The Use of Proofs-as-Programs to Build an Analogy-Based Functional Program Editor

Jonathan N.D. Whittle

Ph.D.
University of Edinburgh
1999

Abstract

This thesis presents a novel application of the technique known as proofs-as-programs. Proofs-as-programs defines a correspondence between proofs in a constructive logic and functional programs. By using this correspondence, a functional program may be represented directly as the proof of a specification and so the program may be analysed within this proof framework. *CYNTHIA* is a program editor for the functional language ML which uses proofs-as-programs to analyse users' programs as they are written. So that the user requires no knowledge of proof theory, the underlying proof representation is completely hidden.

The proof framework allows programs written in *CYNTHIA* to be checked to be syntactically correct, well-typed, well-defined and terminating.

CYNTHIA also embodies the idea of programming by analogy — rather than starting from scratch, users always begin with an existing function definition. They then apply a sequence of high-level editing commands which transform this starting definition into the one required. These commands preserve correctness and also increase programming efficiency by automating commonly occurring steps.

The design and implementation of *CYNTHIA* is described and its role as a novice programming environment is investigated. Use by experts is possible but only a subset of ML is currently supported. Two major trials of *CYNTHIA* have shown that *CYNTHIA* is well-suited as a teaching tool. Users of *CYNTHIA* make fewer programming errors and the feedback facilities of *CYNTHIA* mean that it is easier to track down the source of errors when they do occur.

Acknowledgements

My principal supervisor Alan Bundy has been an inspiration throughout the duration of this thesis. His ability to provide just the right amount of prodding makes him a valuable supervisor. My auxiliary supervisors Richard Boulton and Helen Lowe complement each other perfectly. Richard provided endless technical expertise whilst Helen's knowledge and enthusiasm for the evaluation side of things was encouraging.

A special thanks must go to Andrew Cumming and his students. Andrew allowed me to try out *CYNTHIA* on not one but two of his classes. Beyond this, he showed an interest in how the evaluations should be carried out and helped with the organisation at Napier.

I also thank the following. David Duncan for help with Tcl/Tk. Paul Brna for comments on papers and draft chapters. Alex Blewitt for reading a draft of the thesis. The DREAMers for continuing to dream.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research.

J.N.D. Whittle
Edinburgh
January 29, 1999

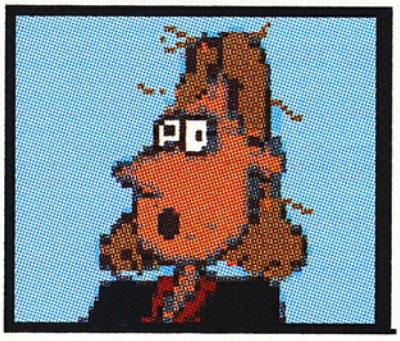
Publications

Some of the work presented in this thesis has previously been published:

- An Editor for Helping Novices to Learn Standard ML. Whittle, J., Bundy, A. and Lowe, H. *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP)*, 1997.
- Supporting Programming by Analogy in the Learning of Functional Programming. Whittle, J., Bundy, A. and Lowe, H. *Proceedings of the 8th International Conference on Artificial Intelligence in Education (AIED)*, 1997.

Contents

List of Figures


PO

TikaLand is a place of dense forests with narrow, winding and sometimes broken paths that wend their way through the thicket. Although the land has been charted many times, the navigation details are kept secret from the outside world. The only clues to the explorer are infrequent signs that give the traveller cryptic clues on where to go next. Unfortunately, these signs are often misleading and confusing.

Mavis arrived in this hostile land by ship. The jetty was at one corner of the peninsula in perhaps the only wide open space before the forest began to the north. The ship's captain had promised Mavis a guide for her journey, but such guides, being in high demand, had already been approached by other passengers. She could still make out the last of them disappearing into the distance. "Oh well," Mavis thought. "At least there're the signs." And she picked up her backpack and headed into the woods.

Chapter 1

Introduction

Formal methods "use mathematical notation and logical reasoning to describe the desired behaviour of a system and to ensure that the final implementation satisfies that behaviour" [Casey 94]. The dream is that by applying formal methods to a programming task, we will be able to reduce the number of errors that the code contains and hence be more confident that our programs do what they are supposed to do. In practice, however, the techniques of formal methods have not been fully embraced by the industrial community. Formal methods are invariably costly (in time and skill levels) and can be difficult to understand. Systems implementing formal methods require designers with a high level of expertise and a grounding in the notions of formal specification and proof.

This thesis presents a novel application of a particular formal method, *proofs-as-programs*. It investigates what kind of program analyses are most useful to the working programmer. The concrete result of this thesis is $CYNTHIA$, a program editor which uses proofs-as-programs as a framework for the automatic, incremental analysis of programs written using it. $CYNTHIA$ is meant to be used for developing real programs by users who have no knowledge of logic and proof. As such, the formal methods have to be hidden from the user. The user is unaware of the analysis going on behind the scenes and does not interact directly with the program analysis modules. $CYNTHIA$ is therefore an example of a system utilising formal methods in a real, practical way that requires the user to know nothing about the formal methods themselves.

$CYNTHIA$ is a programming environment for the functional programming language

ML. Functional programming is a type of declarative programming where the user writes mathematical functions. Each function definition is essentially a collection of equations that are true and complete. Functional programs can be analysed in a direct way using formal methods. ML was originally developed as the Meta Language for the programming of proof strategies in Edinburgh LCF [Gordon *et al.* 79]. It is a typed language incorporating extensive use of pattern matching, recursion and type inference. Although many dialects of ML exist, the most common is the standardised version Standard ML [Milner *et al.* 90].

At present, most users of ML write programs in a text editor and then compile their code with one of the available compilers. *CYNTHIA* provides an alternative platform for writing ML programs. Unlike the traditional approach, *CYNTHIA* implements two original features. Firstly, all programs written in *CYNTHIA* are analysed as they are written. As soon as any errors are introduced into the program, the user is notified and in some cases, the exact location of the error is highlighted. *CYNTHIA* programs are analysed with respect to the following kinds of correctness:

- **Syntactic correctness.** *CYNTHIA* is a hybrid between a structure editor and a text editor. Structure editing inserts syntactically correct code fragments into a program. All freely entered text is parsed immediately and is only accepted if devoid of syntax errors.

- **Type correctness.** If the user introduces a type inconsistency into a program, *CYNTHIA* will highlight the type error. All type errors must be removed before the program can be compiled.

- **Well-definedness.** A well-defined function definition is one that is neither over- nor under-defined. A function definition is over-defined if it is one-to-many. It is under-defined if there exists an element in the domain of the function that does not have an output (or image). Ill-defined functions may be the source of run-time errors. All functions in *CYNTHIA* are guaranteed to be well-defined.

- **Static Semantic correctness**[1]. Static semantic errors are semantic errors that are trapped by a compiler before the program is run. Examples are an undeclared variable or function.

- **Termination**. All programs in *CYNTHIA* are checked to be terminating. Termination checking is an undecidable problem. *CYNTHIA* restricts the user to a decidable subset of terminating programs. This is the set of Walther Recursive programs [McAllester & Arkoudas 96]. This set contains a wide variety of recursive programs sufficient for use in real programming situations, such as multiple recursions, nested recursions, recursion with accumulators etc.

As a theoretical framework in which to analyse these kinds of correctness, *CYNTHIA* incorporates the technique known as proofs-as-programs which says that there is a one-one correspondence between functional terms of the λ-calculus and proofs in a constructive logic [Howard 80]. The λ-calculus can be viewed as a functional programming language, so there is a one-one correspondence between functional programs and constructive logic proofs. This thesis describes how this correspondence has been extended to a subset of ML programs. In *CYNTHIA*, each function definition is represented as a proof. The top-level goal (or specification) of the proof is the top-level type of the function plus bounding lemmas needed for termination analysis. Any proof of this specification provides the correctness guarantees noted above. *CYNTHIA*'s graphical user interface hides all proof details from the user. When the user edits a program, he[2] is in fact, without realising it, editing a proof. At any stage, *CYNTHIA* represents the current program by a proof. Edits made to the program correspond to isolated edits applied to this proof. Any changes made to one part of the proof are propagated throughout the rest of the proof. This proof is then translated into a new program which corresponds directly to a program edit.

The other major difference between *CYNTHIA* and conventional text editors is that *CYNTHIA* provides the user with a collection of editing commands which can be

[1] Type errors and well-definedness errors are also static semantic errors but are distinguished throughout the thesis. The phrase 'static semantic errors' will always exclude type and well-definedness errors. Note also that well-definedness errors are not considered errors by ML compilers but are flagged as warnings. However, it is generally considered good practice to make functions well-defined so ill-definedness is counted as incorrect in this thesis.

[2] Throughout this thesis, I will use 'he' as a pronoun representing either a male or female.

used to transform old definitions into new ones. The idea is that rather than starting from scratch the user chooses a previously defined function which is close to the one which he now wants to define. He then applies a sequence of editing commands to transform his starting function into the one required. The editing commands preserve the correctness at each stage. This concept is called *programming by analogy.* Some of the editing commands are similar to those found in traditional structure editors — e.g. inserting a template for a program construct. However, most of the commands are global commands that do more than make isolated, tiny adjustments. The CHANGE TYPE command, for instance, can be used to change part of the top-level type of a function and watch the effects of this propagate throughout the entire function definition. The proofs-as-programs framework is ideal for this kind of propagation as all that needs to be done is to replay the inference rules that formed the starting proof.

1.1 Example

This section presents a brief working example to illustrate a typical interaction with *CYNTHIA.* Suppose the user wants to write a function, `count`, that counts the number of leaf nodes in a binary tree. Suppose he has already written a function, `length`, that counts the number of elements in a list. Since the two functions are structurally similar, he decides to use `length` as a starting point and transform it into `count`. `length` is given in Figure 1.1 and is defined as a recursive function. The intricacies of the ML language will be deferred until Chapter 2. For now, it is enough to know that `'a list` is the *polymorphic* list type (i.e. the type of the elements of the list is unspecified, but all elements must have the same type), that `nil` is the empty list and that `x::xs` is a list formed by consing the element `x` onto the list `xs`. The first thing to do is to rename the function. By clicking on any occurrence of `length`, a menu pops up displaying all the editing commands applicable at this point of the program. The user selects RENAME and is prompted with a dialog box in which he enters the new name, `count`. *CYNTHIA* then automatically changes all other occurrences of `length` to `count` as in Figure 1.2.

`count` should count nodes in a tree so we need to change the type of the input. To do this, the user clicks on the input type, `'a list`, selects the command CHANGE TYPE

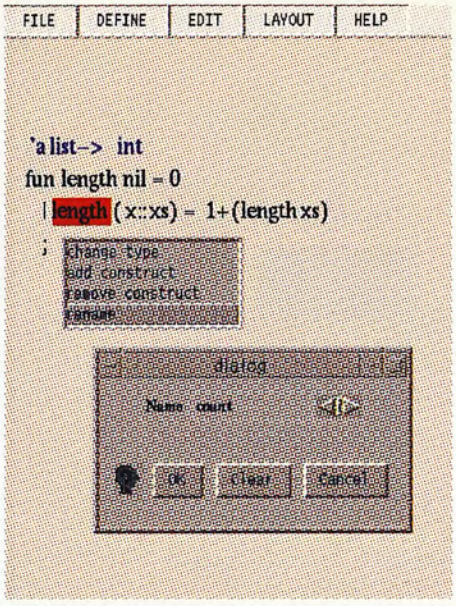


Figure 1.1: Renaming the `length` function.

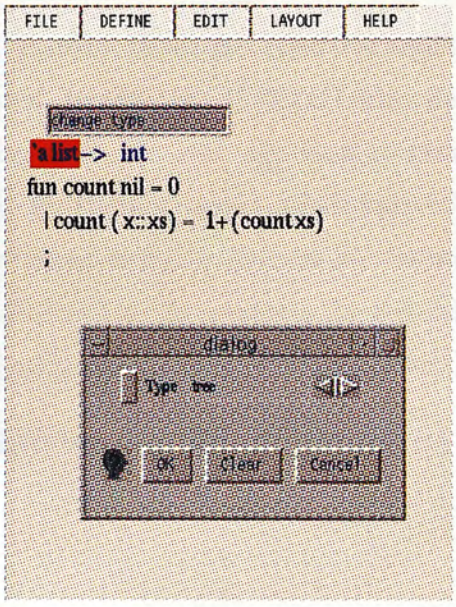


Figure 1.2: A change of type.

and enters the new datatype, `tree` (assume this is already defined). *CYNTHIA* then propagates this change by changing `nil` to `leaf e0` and `x::xs` to `node(xs,e1)` where `e0` and `e1` are fresh variables (Figure 1.3). Note how *CYNTHIA* has automatically produced a well-defined pattern for the `tree` datatype. This pattern consists of two clauses — one for each constructor of `tree`. Note also that the variable `x` has disappeared — this is because `x` is a non-recursive argument of `::` whereas `node` has no

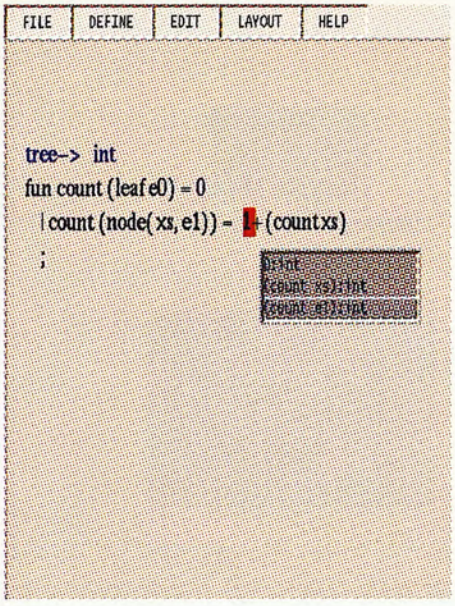


Figure 1.3: Changing the output (1).

non-recursive arguments. Instead, it has two recursive arguments so a fresh recursive parameter, `e1`, is introduced. In addition, *CYNTHIA* has deduced that because `node` has an extra recursive argument, the user may want to make use of a new recursive call with `e1` as parameter. By clicking on the middle button, this new recursive call is displayed. To change the output in the second clause, the user selects `count e1` from this menu. The result is Figure 1.4. To complete the definition, the user changes the output of the first clause using the CHANGE TERM command.

At all stages of the editing process, the definition is correct (in the sense defined earlier).

1.2 Results

CYNTHIA has been fully designed and implemented and is a working editor that supports a functional subset of the Core[3] ML language. *CYNTHIA* has been tested extensively at Napier University and also at the University of Edinburgh. In total, *CYNTHIA* has been used by over 60 people. Ultimately, I believe that an editor such as *CYNTHIA* would be useful to both expert and novice users. As a first step, however, *CYNTHIA* has been designed with novice users in mind. This obviously had

[3] The Core language excludes the ML module system

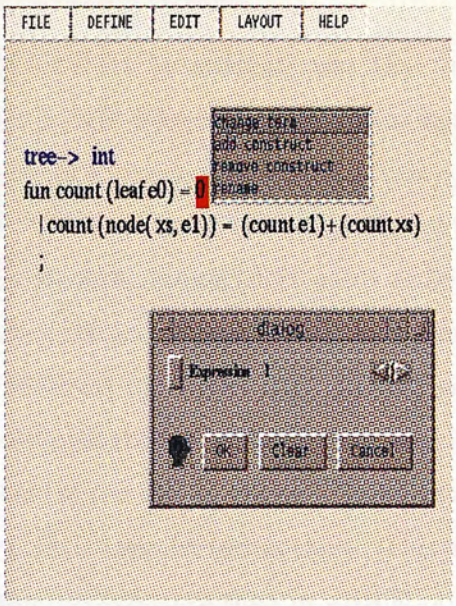


Figure 1.4: Changing the output (2).

some effect on the final look of *CYNTHIA* as well as providing a rich opportunity for studying functional programming students to determine the most common difficulties that novices encounter when programming functionally. Experience with *CYNTHIA* has shown that it is ideally suited for novices. The idea of programming by analogy provides convenient guidance for the student and overcomes the "blank page" problem whereby students just do not know where to start. The advanced correctness-analysing facilities for *CYNTHIA* mean that students generally make fewer errors and in addition, when errors are made *CYNTHIA* provides helpful guidance on the location of the errors. This is in stark contrast to the traditional way of using a text editor and compiler to write functional programs. Current compilers provide very cryptic error messages which students find extremely difficult to decipher. The inherent complexity of the type inference mechanism in ML means that such problems cannot be solved by merely re-designing the compiler error feedback. Fewer errors are made in *CYNTHIA* for a variety of reasons. First, the editing commands often automate programming steps and introduce program fragments which are *a priori* guaranteed correct. This means that users commit fewer errors in the first place. Second, the improved error feedback facilities, such as highlighting the source of type errors, means that less time is spent in finding the location where an error occurred.

The main criticism of *CYNTHIA* by students was that its interface is awkward. *CYNTHIA*'s interface is a hybrid between a text editor and a structure editor but firstly, the text editor is very primitive not including advanced facilities of modern editors like Emacs [Stallman 79] and secondly, the integration of the structure editing and the text editing is not very smooth. Major effort needs to be invested in re-designing the interface, thus increasing its acceptance among its users. This is especially important for extending *CYNTHIA* for expert use. As well as a new interface, such a version of *CYNTHIA* would probably incorporate more specialised editing commands and would support the whole of the ML language. The effects of *CYNTHIA* could be maximised by integrating it with compiler technologies. This might lead to *CYNTHIA* being used more for program maintenance than in writing programs in the first place. It seems that *CYNTHIA* could be a powerful and useful tool in this domain.

1.3 Contributions of this Thesis

There are two major contributions made by this thesis.

1.3.1 Application of Proofs-as-Programs

The proofs-as-programs idea is not new but it has not previously been used as part of a programming environment. *CYNTHIA* uses proofs-as-programs to analyse ML programs but in a way that the underlying proof framework is hidden from the user. Other systems using formal methods require a great deal of expertise to use them. The application of proofs-as-programs involved:

- Developing inference rules to represent ML programs;
- developing an analogical mechanism for propagating user's edits throughout the proofs;
- using incomplete proof information to provide direct feedback to the user about program errors;
- extending Walther Recursion [McAllester & Arkoudas 96] to the ML language;

- designing an interface to hide the underlying proof framework;
- developing an algorithm for automatically carrying out complex changes of type such as given in the example earlier in this chapter.

1.3.2 A Novel Programming Editor

CYNTHIA is an original, state-of-the-art ML editor. It provides new features as well as improving on features that other editors possess:

- Programming by analogy where users adapt existing programs using editing commands;
- a compact, small set of editing commands for program transformation;
- a termination checker plus other non-trivial kinds of program analysis;
- improved type error feedback;
- a structure editor.

1.4 Layout of this Thesis

Chapter 2 presents the ML language and describes the main problems that newcomers to ML (and more generally functional programming) encounter. The discussion is based around results in the literature and empirical studies of novice ML programmers.

Chapter 3 is a literature survey of the area. Systems that do a similar kind of correctness analysis as well as other pedagogic environments are presented.

Chapter 4 gives an overview of *CYNTHIA*. The editing commands are defined from the point of view of the programmer.

Chapter 5 gives the necessary technical background for later chapters. The proofs-as-programs technique is described in detail.

Chapter 6 gives the underlying theory of *CYNTHIA*. It exhibits an encoding of ML programs as constructive logic proofs. In addition, the chapter includes a description of how the proof is updated when editing commands are applied.

Chapter 7 describes the editing commands from a technical point of view. The more complex editing commands are explained in detail. A proof of correctness of the CHANGE TYPE command is given.

Chapter 8 describes two empirical evaluations of the *CYNTHIA* system undertaken on groups of novice ML students at Napier University.

Chapter 9 examines related systems and frameworks. This includes the use of attribute grammars to maintain semantic consistency in programming environments as well as the state-of-the-art ML programming editors.

Chapter 10 outlines areas of future development of *CYNTHIA*.

Chapter 11 summarises the thesis and draws conclusions.

Appendix A includes material relevant to the evaluations in Chapter 8 — questionnaires, error classification tables, examples used in experiments etc.

Appendix B gives a tutorial and manual for *CYNTHIA* as provided to the students that participated in the evaluations.

Appendix C gives the questionnaire used in the experiment in Chapter 2.

Appendix D provides a grammar of the subset of ML supported by *CYNTHIA*.

Appendix E provides a list of the SML of New Jersey system messages that are referred to in Chapter 2.

Mavis had heard in fairy tales of trees mysteriously closing in on someone so that their point of entry could no longer be seen. Mavis would never have believed such stories before but within minutes of entering the forest, she hadn't a clue where she was. She seemed to be at some sort of junction with countless paths going off in all directions. Each path had its own sign but this only added to the confusion. "This road never ends," said one. "Not everyone can pass through here," proclaimed another. "This path doubles back on itself", "This path has potholes", "This path takes a long time", "This path isn't a path". And so it went on until the grand finale: "This path isn't like any path you've seen before"!

Mavis dropped her backpack and slumped onto it in despair. "I need help," she sighed.

Chapter 2

Problems with Functional Programming

2.1 Introduction

The vast majority of programmers are still introduced to the world of computers through an imperative language such as BASIC, C or PASCAL. Functional programming (FP) languages are gaining in popularity but are mostly taught in universities and colleges as second languages. This is beginning to change, however, as FP languages are widely regarded as suitable vehicles for teaching good programming skills such as abstraction, composing programs from smaller units and writing clear, legible code [Davison 94]. Claims have been made that FP languages are ideal as an introductory language [Hudak *et al.* 96]. The absence of side effects in pure functional programming leads to modular programs that are relatively easy to compose and are well-structured, while generally containing fewer bugs than imperative programs. The sound mathematical basis of many FP languages also yields a framework for reasoning about functional programs, leading to the possibility of formally verifying that programs satisfy a specification or automatically transforming inefficient programs into more efficient ones.

However, despite all of these obvious advantages, novices find FP languages notoriously difficult to learn [Thompson 97, Burton 95]. A large body of research has illustrated that students transferring from imperative to FP (or more generally declarative) languages become easily confused when confronted with the significantly different, yet

conceptually simpler, style of FP. Although the FP paradigm is ultimately cleaner, much more effort is needed to acquire the particular programming style. If novices do not grasp this style early on, they can find the going tough.

The situation is made worse by the fact that there exist at present very few good programming environments for FP languages. Most tools provide only a command-line interface and crude error feedback mechanisms that provide only very limited help to the user in locating errors. In contrast, the widespread commercial use of more traditional languages such as C and BASIC has meant that sophisticated tools have been provided to help the user as much as possible.

The main argument presented in this chapter is that current environments for FP languages are inadequate to maximise the learning potential of FP novices. I identify the key areas which cause difficulties to students and describe how these problems might be overcome by using an editor based on techniques from the field of automated reasoning. The aim is not to produce a tutoring system but instead to provide intelligent support for the novice when programming functionally, in terms of reducing the number of programming errors that the user makes and providing more useful feedback when bugs do occur.

This chapter first gives a brief overview of the FP language ML since the end-product of this research uses ML as a case study. I identify the main problem areas when learning ML but note that these apply more generally to any FP language, except where otherwise stated. The discussion motivates the development of an analogy-based editor for ML, named *CYNTHIA*. The techniques that *CYNTHIA* embodies are briefly described here but for a more technical description the reader is referred to later chapters.

The arguments provided here are based upon results in the available literature and also on my own knowledge of novice functional programming. This consists of teaching experience, having tutored three ML courses and two LISP courses, and also empirical studies of novice ML programmers at Napier University in Edinburgh. In total, three groups of students at Napier were studied. This chapter reports on the first of these. The study was undertaken during September – December, 1996, that is before the

main design phase of *CYNTHIA* was embarked upon. The results of the experiments directly affected the design of the editor. The subjects in the experiment were post-graduate students following an MSc Postgraduate Diploma in Software Technology at Napier University. This course is designed for those with some experience in computing. The students came from a variety of backgrounds — some having worked in industry, some having come straight from an undergraduate course. Many had programmed in non-functional languages such as BASIC, PASCAL or C. None of the students had used ML before. 14 students were involved in the study. Table 2.1 shows the computing experience of the students following the course. A tick in the relevant box indicates that the student has taken at least one course in this language.

The ML course was designed and lectured by Andrew Cumming. Each student was assigned one supervised and one unsupervised session per week, each lasting two hours, in which they worked at examples from the course literature. The course notes are available on the WWW [GIML]. They are divided into eight sections each introducing an important ML concept. Each section gives a short tutorial introduction to the concept and then presents the student with a series of exercises that they are expected to attempt in the laboratory sessions. The exercises typically involve writing small (a few lines) functional programs but occasionally "diversions" are given which lead the user through a more complex programming task. In the early part of the course, the students typically cut and paste problems from the notes. Later, they write definitions using a text editor and compile their programs. The compiler used was version 0.93 of Standard ML of New Jersey (SML-NJ).

Student	Pascal	SQL	COBOL	C/C++	Oracle	BASIC	LISP	Fortran
1	√	√						
2	√		√	√	√			
3				√	√	√		
4	√	√	√	√		√	√	
5				√				√
6								
7								
8	√			√				
9								
10	√					√		
11								
12								
13								
14								

Table 2.1: Student Programming Experience.

To give the reader some idea of the scope of this particular course, I will give an overview of the concepts introduced at each stage. Firstly, however, it is necessary to acquaint the reader with ML.

2.2 Brief Introduction to ML

There are a number of different dialects of ML. Throughout the rest of this thesis, I confine attention to the recognised dialect Standard ML which is the most widely used. I do not attempt to describe functional programming here but refer the unfamiliar reader to [Bird & Wadler 88]. ML was originally designed for programming proof strategies in Edinburgh LCF [Gordon *et al.* 79]. Since then, however, it has grown and been used in a wide variety of applications. The syntax of ML is best illustrated by example. The following program is a piece of ML code to calculate the length of a list:

```
fun length nil = 0
|   length (h::t) = 1 + length t;
```

where `nil` denotes the empty list and `::` is the *cons* operator for constructing lists. Note the use of recursion as is common in functional programming. The main features that distinguish ML are as follows:

- **Higher order programming**. This means that ML functions can take other functions as arguments. In this case, the function is a *higher-order* function or *functional*. Although procedural languages such as Fortran may allow functions to be passed as arguments to another function, few procedural languages allow function values to play a full role in data structures [Paulson 91].

- **ML is strongly typed**. Every object in the language belongs to a type such as list, integer or tree. ML employs *type inference* to automatically infer types at compile time. This frees the user from declaring types in most cases. For example, an ML compiler would deduce that `length` has type
  `'a list -> int`. `int` is the type of integers. `'a list` is a *polymorphic* list — i.e. the type of elements of the list is unspecified. Polymorphism allows code to be shared between different data structures while retaining the security associated with strong typing.

- **Modular programming** allows the structuring of a large program as a system of simple parts, connected by simple interfaces [Paulson 91]. It includes the notion of *abstract types* or types whose internal representation is hidden so that the representation may be changed without affecting programs defined on the type. The subset of Standard ML excluding modules is sometimes referred to as *Core* ML.[1]

- **ML is strict.** This means that to compute the value of $f(E)$, first compute the value of E. This is in contrast to *lazy* languages which only compute the arguments of a function as necessary. Given the function:

  ```
  fun f x = 0;
  ```

  ML would always compute the value of `x` before returning `0` as the result of `f x`. Lazy languages, such as Haskell [Hudak *et al.* 96] or Miranda [Holyer 91], realise that the computation of `x` is unnecessary and return `0` immediately. There are pros and cons for using strict rather than lazy evaluation. See [Paulson 91, p.44] for a discussion.

- **ML is an impure functional language** — i.e. it contains some imperative features included for practical reasons. The main use of imperative programming is in input / output.

- **ML has exceptions** — exceptions are a way of leaving a result undefined. An example is the following:

  ```
  'a list -> 'a
  fun listhead nil = raise excep
  |   listhead (h::t) = h;
  ```

CYNTHIA is concerned only with a purely functional subset of Core ML (defined explicitly in Appendix D) (although *CYNTHIA* does partially support exceptions). This is on the grounds that purely functional definitions are easier to analyse and also that the module system is less likely to be used by novices than by expert programmers. Further particulars of ML will be explained as they are introduced. For a fuller description of ML, I refer the reader to one of the many books on the subject — [Paulson 91, Michaelson 95, Ullman 94].

[1] Note, however, that one can define abstract types in Core ML using the `abstype` construct.

2.3 ML at Napier

This section gives a brief overview of the ML course at Napier University. As previously mentioned, the course material is split into eight sections. These sections consist of material as follows:

- **Lesson 1**: Expressions and simple functions.
- **Lesson 2**: Types, bindings, pattern matching (see §2.4.2) and lists.
- **Lesson 3**: Simple recursion on integers.
- **Lesson 4**: List processing including recursion.
- **Lesson 5**: Partial functions, overlapping patterns, anonymous functions, more complex recursion.
- **Lesson 6**: Higher-order programming.
- **Lesson 7**: User-defined types.
- **Lesson 8**: Accumulators in recursion, mutual recursion, nested definitions.

2.4 Why FP languages are difficult to learn

2.4.1 Functional Programming Style

In order to maximise the potential of an editor such as *CYNTHIA*, it is necessary to identify the key problem areas of functional programming. The problems can be divided into two parts — those specific to ML and those common to all declarative languages. As far as the latter goes, most research into novices' misconceptions has examined the logic programming language Prolog (see [Clocksin & Mellish 81] for an introduction to Prolog). Although many of the results in the literature are specific to Prolog, just as many are relevant to all forms of declarative programming [Taylor & duBoulay 87, Taylor 88, Pain & Bundy 87]. I will only give a brief summary of such problems here for it is not the purpose of *CYNTHIA* to deal with high-level, deeply conceptual programming problems. Rather, it is the intention to

use *CYNTHIA* to filter out some of the more obvious (although still interesting) programming errors so that the student's time is freed to deal with the deeper difficulties. However, I introduce the deeper misconceptions here to illustrate how students must spend a good proportion of their time dealing with such issues.

Many misconceptions arise due to the fact that students have pre-conceived ideas learnt from procedural programming that they try to apply, unsuccessfully, to the declarative world. The problems to be described also occur, but to a lesser extent, in those students with no knowledge of programming. The language of expression in declarative programming languages is often foreign to students. In FP languages, it tends to be mathematical and it has been suggested that students with a weak scientific or mathematical background will find FP languages harder to master. The higher-level nature of the languages can be problematic. This can lead the student to believe that less rigour is needed when programming declaratively. Logical and functional expressions can be easily misconstrued unless a genuine understanding of the underlying semantics of the language is achieved. Related to this point is the fact that there is often a lot going on "behind the scenes" in declarative languages as the control flow of the program is pre-determined. Whilst this is more of a problem in Prolog which necessitates an understanding of backtracking, the problem is still evident in FP languages. Procedural languages give the user complete control of the program and so there are fewer unexpected program behaviours. In declarative languages, the problem solving strategies are usually very different. The student must learn to use the underlying control flow to its best effect. Indeed, the actual problems that the student will tackle are likely to be vastly different as well. For some languages, the domain of discourse is well defined (e.g. numerical). Declarative languages are not particularly well suited to number crunching. Instead, the domain can be objects that the student has not necessarily considered suitable for programming before. This leads to a problem of representation. Declarative languages offer a greater degree of freedom of expression [Taylor & duBoulay 87] and this can often hinder rather than help the novice.

One more pragmatic difficulty is to do with the way FP languages are usually presented. Functional programs are a collection of definitions. Usually, each definition performs

some small task and large tasks can be undertaken by combining a set of definitions. Novice students often have a great deal of difficulty in conceptualising what a large program will look like. Their first couple of sessions with ML will introduce them to small functions such as a function to double a given number, but they will have very limited understanding of how to combine functions together into a larger program. These introductory functions often seem to perform only rudimentary tasks and it can be quite hard for novices to imagine a really useful program written in ML. This can lead to a mistrust of the language. In fact, declarative languages are often deeply unpopular. One student at the University of Edinburgh described ML as "the bane of our lives". ML is viewed as being an inherently mathematical language. As a result, many students with a lack of a mathematical background feel ill-equipped to ever be successful with ML. They are, in their eyes, disadvantaged from the start and this sort of negative feeling will undoubtedly mean that their progress will be adversely affected. A common reaction is that the tasks could have been done better in other languages anyway.

The most common way for programmers to learn a language is to sit down and to try programs out. They enter a compile-edit-compile cycle where they debug or extend their program in response to the system's responses. This is an inefficient way to program. The approach will invariably produce code that is difficult to read and understand. Unfortunately, however, it is an arduous task to change the way that novices think about programming. Ideally, they should design and plan a solution before attempting to implement it, but this process takes more time in the short-term. Novices have to be persuaded of the long-term benefits of such an approach. This leads to a battle between what they are told is good programming practice and the form of hacking-as-programming that seems more natural and produces immediate 'results'. ML by its very nature actively encourages a good programming style. As a result, students associate it with an approach to programming that negates all their previous programming experiences. ML's very design is forcing them to adopt an approach that they have managed for years without. It is factors like these that lead to the problems of transfer from procedural to functional languages.

Whether the above arguments are due merely to FP's small share of the program-

ming language pie, or whether the style of functional programming is inherently more counter-intuitive is unclear. Whichever is true, there are clearly some major conceptual hurdles to be overcome by the programmer new to FP languages.

2.4.2 Functional Programming Concepts

There are some more specific barriers to be overcome when it comes to learning FP languages. This section outlines what they are and why they cause problems.

Recursion

Recursion is well-known to be a difficult concept to learn [Anderson *et al.* 88]. Although languages such as C do support recursive definitions, the more usual approach is to use iteration which is often said to be more natural. There are few everyday analogies for recursion [Bhuiyan *et al.* 94]. In many FP languages and also in logic programming languages such as Prolog [Clocksin & Mellish 81], recursion is the main way of making definitions. As [Bhuiyan *et al.* 94] states:

> *The difficulty in developing recursive solutions is due to learners' inability to express a solution recursively and their inability to understand the suspended computation, and thus the unravelling of recursive calls.*

Many approaches have been undertaken to attempt to explain recursion to novices in a way that they will understand. These range from providing different representations of recursion to relating recursion to the more easily understood concept of iteration. Visualisation systems such as [Sutinen *et al.* 97] give a graphical representation of a recursive algorithm. This representation may be static or it may be dynamic whereby the algorithm is effectively executed and changes in the display image are seen as the algorithm progresses. Early approaches of this kind relied on pre-stored animations and hence were not interactive. Later systems, such as [Haajanen *et al.* 97], allow the user to interactively create their own animations and hence interact with the recursive procedures. An alternative approach to visualisation is provided by systems such as PETAL [Bhuiyan *et al.* 94] which implement different ways of thinking about recursion. In PETAL, recursive definitions can be made from various viewpoints. One

viewpoint is to see recursion as filling in the base and step case(s) in a template of a recursive algorithm. Another is to analyze the input-output behaviour in more abstract terms that do not rely on particular code fragments. [Pirolli & Anderson 85] suggests that the easiest way to learn recursion is to study and modify existing examples. In fact, there has been a large body of research that provide ample evidence that a good way to help a novice learn recursion is to make analogies with previous, correctly defined recursive procedures.

It is precisely this analogical approach that is used in *CYNTHIA*. As mentioned in the previous chapter, programs are written in *CYNTHIA* by editing existing programs. By using this approach, the user is encouraged to make analogies between different tasks and this should help him identify the structural properties of each particular task, in particular with regard to the kind of recursion in the program. Note, however, that *CYNTHIA* is much more flexible than systems such as PETAL as the user is not restricted to a pre-defined set of examples but can edit existing programs to an arbitrary degree of complexity.

My own investigations show further evidence that analogy is used as a problem-solving method by students. When confronted with a new task, students invariably were seen to make connections with examples they had previously defined or that were available in the course notes. At the simplest levels, this might be to realise that to sum the elements in a list it is necessary only to change a tiny part of the `length` function. At deeper levels, students were seen to think initially about what type of recursion scheme was required. The approach taken in *CYNTHIA* is to encourage this kind of reasoning by providing a starting point for the student. Our approach is more general than just providing a recursive template as, firstly, an entire function definition is provided (i.e. not a template with gaps to fill) and, secondly, because unlike with conventional template-based approaches the user is not restricted to use the recursion scheme he is provided with. If the user realises part way through the editing process that in fact a different recursion scheme is needed, he can apply editing commands to construct this recursion scheme without having to start again.

More will be said about this approach in §2.6. It is worth mentioning here though a couple of studies that claim that analogy can be a flawed problem-solving technique.

[Escott & McCalla 88] found that students can make programming errors if they rely on analogy because they do not make all the changes necessary to produce the new program. Shirley Booth[2] claims that reasoning by analogy is in fact an expert technique and should not be taught initially to novices because then novices will not pass through the mental processes necessary to make the jump from novice to expert. She claims that novices look only at surface analogies and hence are in danger of making incorrect analogies. The defence to this is that firstly, *CYNTHIA* is not presented as a tool to make analogies. The idea is not that the students will be told to start off with a program and make one or two minor changes to produce a new program. This approach would only be suitable for a very restricted tutoring system. Instead, the user has complete freedom to choose the starting point and go wherever he wants from there with no pre-supposition that the source and target will in fact be at all similar. Of course, the idea is that the novice chooses a similar source program to the one he intends but there is no obligation to do so and hence no tendency to make a small number of minor changes without properly analysing the code. Second, the analogies are generally very simple ones. Because of the nature of functional programming, programs are written bit by bit. In *CYNTHIA*, editing is done for each individual function not for an entire, complex program so the scope for making incorrect analogies is vastly reduced. As for making surface analogies, students were observed to be making exactly the right kind of analogies for an editor such as *CYNTHIA* to succeed. In the context of ML programs, the structural similarities of programs are most important — namely the type of the function definition, the parameter that the recursion is over, the type of this parameter, the kind of recursion scheme. This issue will be discussed further in Chapter 8.

The other aspect of this analogical approach is that it is intended to reduce the number of programming errors. Because the editing commands are correct, and the source program is correct, the target is more likely to be correct. Moreover, because editing is done by applying commands, if an error does appear in the target the user knows that it is the most recently applied edit that introduced this error and hence will have a much improved perception of the cause of the error.

---

[2] personal communication

Two particular kinds of error crop up frequently when dealing with recursive programs. ML programs are defined by pattern matching — for example, the `length` function given earlier was defined for the pattern when the list was `nil` and for when it was `h::t`, i.e. non-empty. The notion can be extended to any ML data structure. It is easy to define partial functions in ML — just delete the case for `nil` for instance — and this can lead to run-time programming errors. ML compilers are able to detect if a function is only partially defined and will warn the user if a pattern is not well-defined. By a *well-defined* pattern, I mean one in which:

- There are no overlapping patterns.
- The patterns exhaustively cover the data structure.

In the case of overlapping patterns, ML compilers resolve the ambiguity by considering patterns in a top-bottom, left-right order. Note, however, that this may not be the result that the programmer intended. Although partial functions can occasionally be useful and are therefore signalled only by warnings (as opposed to errors), it is generally considered a good idea to write total functions wherever possible for this will reduce the likelihood of errors being present. Section 2.5 gives empirical results about how often novices write ill-defined functions.

Another error that novices invariably encounter when dealing with recursive procedures is that of non-termination. An example of a non-terminating function is given by:

```
fun gcd (x:int, y:int) = if x=y then x else gcd (x-y, y);
```

Note the result of calling `gcd(2,3)`. Section 2.5 discusses how common non-termination is among novices.

Typing

ML is an implicitly and strongly typed language. This is in contrast to languages such as LISP which is only weakly typed and PASCAL or C which are explicitly typed. Implicit typing is a very useful feature for a programming language to possess. Users need not state the type of objects that they are defining but the compiler will automatically infer the types from the context, usually via a type inference algorithm

based on Hindley-Milner [Damas & Milner 82]. Whilst this gives the user a great deal of freedom and saves him from constantly entering type declarations, it can be a source of confusion for the novice. Firstly, if there are type inconsistencies in the program, the type inference algorithm will be unable to derive a type declaration and so will produce a type error. Unfortunately, due to the inherent complexity of the algorithm, type error messages can be at best confusing and at worst misleading by identifying the error at the wrong point. Even experts often have difficulty understanding the type error messages of current compilers. Note that the problem lies not in the way a particular compiler presents type error information, but instead in the fact that the Hindley-Milner algorithm may not correspond well to how people understand the types of programs [Jun & Michaelson 98]. Type error messages usually merely indicate where the inference algorithm broke down and this point may be far removed from the *actual* source of the error. This situation is made worse in the case of languages like ML because the added advantage of having polymorphic types can make the process of reporting type errors even more complicated [Beaven & Stansifer 93]. In this case, errors can occur when unification fails between two generic types. Even if type error messages are ignored, novices can find it notoriously difficult to pinpoint the source of an error. Finding the source typically involves unravelling the possibly long chain of deductions and type instantiations made by the compiler. This sort of debugging technique is far from trivial to master. It requires an analytical and mathematical mind. Types can be arbitrarily complex. Novices have enough difficulty dealing with the interaction of the simplest of types but once they encounter anything as complicated as lists of lists, they can become lost very easily. Students can also be confused when the compiler infers a polymorphic type for a function which they intended to be defined over a specific type declaration.

[Jun & Michaelson 98] gives a short account of experiences of undergraduates' misconceptions with types based on six years of teaching. In addition to the points noted above, [Jun & Michaelson 98] notes that novices have problems with types at an even simpler level. They have a lack of understanding of numeric type inconsistencies. Traditional languages permit mixed mode arithmetic and automatic type coercion which does not encourage students to recognise that integers and reals are two distinct types. Also, students are confused between the way FP languages treat the type of booleans —

where `true` and `false` are values — and their use as control indicators for conditional statements in other languages.

Type inference can lead students to ignore types. Because of the cryptic and lengthy nature of error messages, students are seen to initially spend a long time unsuccessfully deciphering them. Having failed, they resort to blind hacking of their code, changing arbitrary parts of a definition, possibly even parts that were correct to begin with. This problem, which can happen with other kinds of errors as well, leads the student to spend a disproportionate amount of their time debugging their programs, but this debugging process has no logic to it. As a particularly illustrative example, during my observations at Napier, one student tried to write `addtolist` taking an integer list and an integer and returning a second integer list constructed from adding the integer to each element of the input list. His first attempt was:

```
fun addtolist (n,nil) = n
|   addtolist (n,h::t) = h+n :: addtolist t;
```

This is correct apart from a type error in the base case and a missing parameter in the recursive call. However, the student could not spot this and concentrated instead on making changes to the first part of the step case: `h::h+n::addlist t` , `h+n @ addlist t`, `h::n::addlist t`. The student spent the majority of his tutorial hour attempting to correct the error but never did find the source. Students need to be encouraged to make full use of type information. They ought to think clearly about the type of the function and the type of expressions within it. Otherwise, they will waste vast amounts of time debugging unsuccessfully and this will have a severe effect on their confidence and attitude towards the language.

Other Problems

Ironically, one reason why a large number of type error messages appear is that the syntax of ML is very succinct. This means that mistakes that in an ideal world would be caught by the parser have an alternative parse and so are only caught by the type checker. A common example of this happening is when students do not fully understand the difference between curried and uncurried functions. Briefly, functions with more than one argument may be implemented as a curried or uncurried function.

An uncurried function has a single argument which is a tuple. The following definition of add is uncurried:

```
(int * int) -> int
fun add (x,y) = x+y;
```

Note that add takes an `int * int` pair as argument. add also has a curried version[3]:

```
int -> int -> int
fun add x y = x+y;
```

This is a function which takes an integer and returns a function from an integer to an integer. We can give both arguments without using a tuple:

```
add 2 3;
```

Giving one argument results in a "partial evaluation" of the function. For example applying the function add to 2 alone results in a function which adds two to its input. Curried functions can be useful — particularly when supplying functions as parameters to other functions.

Suppose the user enters the following line of code:

```
...f (x::xs) y = ...f (xs,y)...
```

This will result in a type error because f is curried in one location and uncurried in another. The likelihood here is that the student is confused about the syntax of defining a function and has unwittingly provided two different, although equally valid, versions.

Another aspect of ML that novices find difficult is the higher-order nature of the language. Although higher-order programming is not usually introduced until the latter half of the course, by which time the students are familiar with the language, the concept still provides a real challenge. A typical higher-order function given in an introductory course is the foldl function:

```
fun foldl f (e,nil) = e
|   foldl f (e,x::xs) = foldl f (f(e,x), xs);
```

which has type `('a * 'b -> 'a) -> 'a * 'b list -> 'a`. I have noticed students having severe difficulties trying to understand how such functions work. Trying to

[3] -> associates to the right.

derive the type of the function or correct type errors in such situations is an even more difficult task.

The preceding sections have outlined the major difficulties encountered by novice ML users. As can be seen, the obstacles are formidable. We cannot hope to eliminate all of the problems but by concentrating on reducing the number of programming errors and providing better feedback for locating errors, the user will be freed to get to grips with the more difficult concepts in the language. The next section goes into more detail about the experiments undertaken at Napier University.

2.5 Analysis of Errors Encountered by Novice ML Programmers

I undertook an empirical investigation into the kinds of errors that novice ML programmers encounter in a first FP language course. The results of this experiment are reported in this section.

2.5.1 Automatic Logging of Students' Interactions

With their permission, the students interaction with SML-NJ was automatically logged over a two hour tutorial session. This took place in the fourth week of the course which coincided with the introduction of recursion on lists. An analysis was made of these scripts in order to determine what kinds of error were made by the students and with what frequency. The measures taken (for each error category) were as follows:

- Percentage of students encountering the error, E
- Number of times the error was encountered for each particular student, N_s
- Frequency that the error was encountered for each particular student, A_s:

$$\frac{N_s * 100}{\textit{no. commands entered in session}}$$

  This measure is intended to give some idea of the frequency of errors. Rather than choosing to evaluate errors per line of code which could be misleading (includes

error messages and irrelevant aspects such as pressing RETURN fifty times) I have made an analysis based on the number of commands entered (all commands are ended by a semicolon so this is fairly reliably measured).

Syntax Errors	
syntax error	NONfix pattern required
unclosed string	nonfix identifier required
Static Semantic Errors	
unbound variable or constructor	operator not a function
clauses don't all have same number of patterns	duplicate variable in pattern
can't find function symbol	constructor used without argument
Type Errors	
tycon mismatch	overloaded variable
Well-Definedness Errors	
match redundant	match non-exhaustive
uncaught Match exception	

Table 2.2: Some SML-NJ error messages.

Errors were classified according to the ML error message given. I was looking for particular kinds of errors — namely syntax errors, static semantic errors, type errors[4], well-definedness errors and termination errors. I grouped each kind of SML-NJ system error into one of these groups — this is shown in Table 2.2 (termination errors are not shown but see §2.5.2). Note that well-definedness errors are not strictly errors. The ML interpreter will accept, for instance, only partially defined functions. However, it is generally considered good programming practice to write total functions wherever possible. Partial functions can lead to run-time errors. In Table 2.2, I leave out the exact meaning of the error messages but refer the interested reader to Appendix E. I have not included a category for looping errors caused by non-terminating programs. These would appear on the script marked by "Interrupt" where the user has typed Control-C to interrupt an infinite recursion. A discussion of termination errors is given in §2.5.2. Table 2.3 gives the errors in each category and Figure 2.1 represents the information as a histogram.

[4] Type errors are static semantic errors but are given a separate category as they are of particular interest.

Student	Syntax		Static Semantic		Type		Defined	
	N_s	A_s	N_s	A_s	N_s	A_s	N_s	A_s
s_1	8	11	20	27	11	15	0	0
s_2	4	4	10	9	36	33	5	5
s_3	12	36	4	13	3	9	0	0
s_4	2	3	1	2	21	34	2	3
s_5	5	12	33	78	8	19	0	0
s_6	3	6	7	14	7	14	0	0
s_7	0	0	0	0	0	0	0	0
s_8	7	9	5	6	12	15	1	1
s_9	8	12	5	8	12	18	0	0
s_{10}	6	10	12	21	37	63	2	3
s_{11}	6	17	1	2	0	0	7	19
s_{12}	0	0	5	6	5	6	1	1
s_{13}	1	3	1	3	5	15	4	12
s_{14}	8	7	10	10	36	34	17	16
	70		114		193		39	
E	86		93		86		57	

Table 2.3: Errors per student.

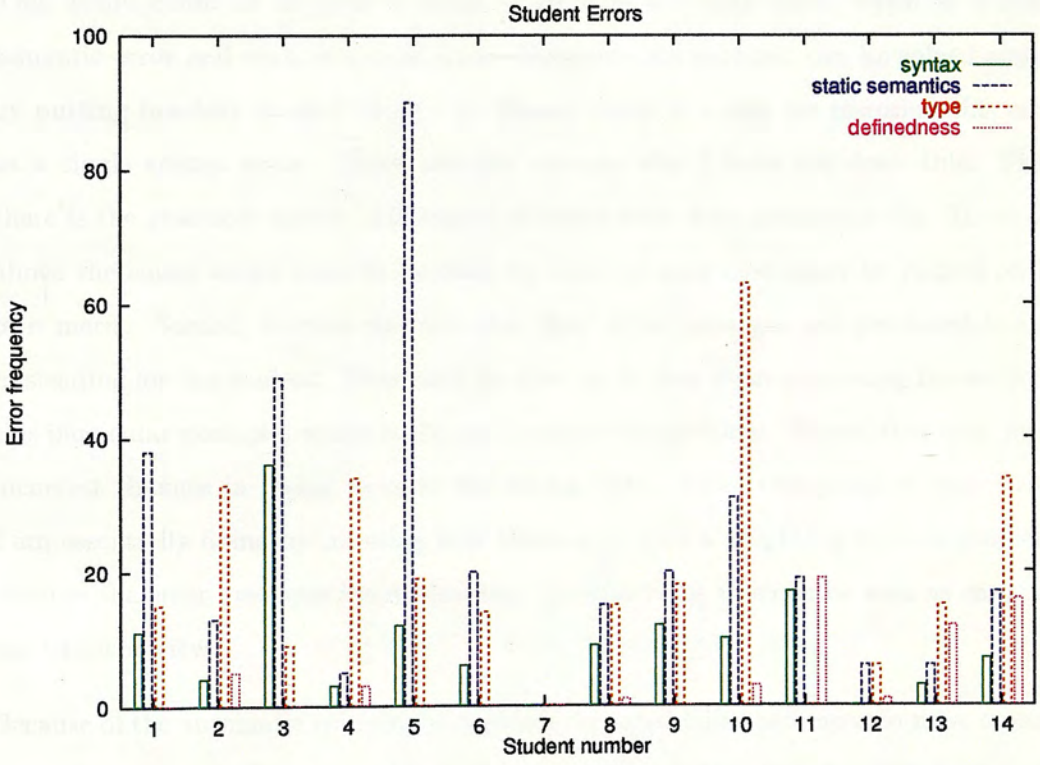


Figure 2.1: Student Errors.

The errors were actually counted automatically. Although this makes the process quick and painless, it can lead to minor problems. Firstly, the various error messages are

not always independent of each other. A particular problem may give rise to other problems which can cause more than one error message. For example, one student scripted:

```
- fun flatten nil = nil
|   flatten (a,b)::t = a :: b :: flatten t;
Error: NONfix pattern required
Error: clauses don't all have same number of patterns
Error: data constructor :: used without argument in pattern
Error: rules don't agree (tycon mismatch)
  expected: 'Z list -> 'Y list
  found:    ('X * 'X) * 'W * 'V -> 'X list
  rule:
   ((a,b),_,t) => :: (a,:: (<exp>,<exp>))
```

This would count as an error 4 times — once as a syntax error, twice as a static semantic error and once as a type error. However, the problem can be solved simply by putting brackets around `(a,b)::t`. Hence, there is a case for counting this error as a single syntax error. There are two reasons why I have not done this. First, there is the practical aspect. All counts of errors were done automatically. To do the above the count would have to be done by hand as each case must be judged on its own merit. Second, I think the fact that four error messages are produced is very misleading for the student. They have no clue (as is clear from examining the script in this particular example) which is the *real* cause of the problem. Hence, they may make incorrect changes in trying to solve the wrong error. From this point of view, what I am essentially doing by counting four times is to give a weighting to such problems because the error messages are misleading. Alternatively, this can be seen as counting the 'student view'.

Because of the automatic counting procedure, repeated failed attempts to solve a problem will count more than once. Again, it is debatable whether this should indeed be the case, but again, I think it is justified because I am essentially weighting the occasions when students find it difficult to uncover the reason for an error.

Of course, the error messages are particular to SML-NJ. I make no claims about other implementations of Standard ML such as MLWorks [Har96] and PolyML [Pol96].

2.5.2 Analysis

From the results of the analyses, it can be seen easily that novice users of ML encounter a large number of errors during their learning process. As expected, syntax errors are high in number. Perhaps unexpectedly, static semantic and type errors occur more frequently. Well-definedness errors, as expected, occur less frequently. These errors are inherently more dangerous, however. Firstly, they can cause run-time errors whereas syntax and static semantic errors will always be caught at compile-time. Secondly, well-definedness errors are more difficult to correct. Although there are odd cases when a student just cannot see the cause of a simple syntax error (e.g. missing space etc.), usually syntax errors are relatively easy to find. Conversely, although well-definedness errors are clearly signalled, it can be quite hard for students to come up with a well-defined set of patterns in non-standard cases. By non-standard I mean when there is more than one pattern for a data constructor — for example, the patterns `nil`, `h::nil`, `h1::h2::t`. In fact, it was often observed that students had real conceptual difficulties in such cases.

The extent to which termination errors occur turned out to be difficult to quantify. As previously mentioned, these kinds of errors can be recognised by the occurrence of 'Interrupt' on the scripts. However, very few such errors were found. Of course, we would expect termination errors to be significantly less frequent than, say type errors but we might expect novices encountering recursion for the first time to write programs with infinite loops more often. I believe there are a number of reasons for the scarcity of termination errors.

- The students were still at an early stage in the course. This meant that the recursive programs they programmed were almost all primitive recursive. The students were less likely to make incorrect recursive calls because the concept of primitive recursion had been stressed in the course material. In addition, most students cut and pasted program text from the course material on the Web page and modified it to suit their particular requirements. This meant that errors were less likely because the programs they pasted into SML were clearly correct. Many tasks involved trivial manipulations of the program that did not affect the

recursive call. I believe that termination would become a bigger problem when students use non-primitive kinds of recursion.

- It was noted during the tutorial sessions that students often became confused when confronted with an infinite loop. Examples were seen where students set off infinite recursions and closed the window to stop the process. This is because ML can slow down considerably when caught in a loop and respond to Control-C very slowly. This meant the script was lost and so suggests that there were more examples of infinite recursions than noted. Moreover, it suggests that infinite recursions were a real obstacle to the student's progress as they had to close the window. Unfortunately, we have little idea how often this situation occurred — although see §2.5.3 on questionnaires which sheds some light.

In addition, termination errors are in many senses more serious than the other kinds of errors. They will only be discovered at run-time and no feedback is provided about why the error occurred.

The quantitative results presented here back up the results of [Bental 95] which reports on experiences with the Ceilidh system at Heriot-Watt University. 60 students used Ceilidh as part of a course on programming in ML. Part of the functionality of Ceilidh allows students to send email to human tutors to ask for help if their programs do not work. [Bental 95] classified these queries according to subject. The results are reproduced in Figure 2.2. Note that the figure only reflects problems of sufficient difficulty that the student could not solve them on their own and had to ask for help. This explains why syntax errors are low in number. The results indicate the relative importance of type errors. [Bental 95] also notes that students had great difficulty understanding the error messages from the ML compiler.

2.5.3 Qualitative Analysis

As well as the quantitative analysis, the students were asked to fill in questionnaires to get some sort of feeling for how they were affected by ML errors. Results for each question are presented in Tables 2.4 and 2.5. The questionnaire can be found in

Syntax of SML	18		
Semantics of SML	53		
		Recursion	19
		Functions as values	13
		Data types	9
		Pattern Matching	5
		Others	7
Type Signatures	12		
Style	18		
Debugging	13		
Interpreting the Question	15		
General Incomprehension of SML	5		

Figure 2.2: Subjects of queries posted by students using the Ceilidh system.

Appendix C. The headings in Tables 2.4 and 2.5 correspond to the question responses in Appendix C. The meaning of 'static semantics' would not be clear to the students so they were only asked about syntax, type and definedness errors. It is expected that most of the static semantic errors were marked as syntax errors by the students. Confronted with a static semantic error such as an unbound variable, the students would be likely to class this as a syntax error.

Error	**Frequency**				**Rectify**					**Encountered**
	N	HE	F	VO	VE	RE	NED	RD	VD	
syntax	0	3	11	0	1	8	4	1	0	14
type	0	4	9	1	1	3	9	1	0	14
loop	9	5	0	0	0	3	0	1	1	5
couldn't identify	3	6	2	3						

Table 2.4: Questionnaire Results — 1.

N : never encountered
HE : hardly ever
F : frequently
VO: very often

VE : very easy to rectify
RE : relatively easy
NED : neither easy nor difficult
RD : relatively difficult
VD : very difficult

2.5.4 Analysis

The results of the questionnaires conform pretty well with the results of the scripting sessions. Syntax and type errors were encountered by a large majority of the students.

Helpfulness of compiler					Debugging Time				
N	S	Mod	Most	E	AN	S	F	L	VL
0	8	4	1	1	0	3	6	5	0

Table 2.5: Questionnaire Results — 2.

N : not at all helpful
S : slightly helpful
Mod : moderately helpful
Most : mostly helpful
E : extremely helpful

AN : almost no time spent debugging
S : a small amount of time
F : a fair amount
L : a large amount
VL : a very large amount

The results for syntax and type errors are similar. In both cases, most students came across these types of errors frequently. It seems, however, that the students found type errors more difficult to correct — 8 students found syntax errors relatively easy to correct and 9 students found type errors neither easy nor difficult to correct.

In hindsight, it would have been useful to ask about well-definedness errors in the questionnaires. As I did not do this, no information is available for comparison with the scripting. As for termination errors, 5 students encountered these which gives credence to the speculations made earlier about why termination errors are generally absent from the scripts.

Section 2 of the questionnaire was intended to survey the students overall feeling about the helpfulness of the SML-NJ compiler in correcting errors. The majority of students found the compiler only slightly helpful. Only 2 out of 14 thought it offered more than moderate assistance. A good proportion of students spent a large amount of their time on correcting errors. The majority (11 out of 14) spent at least a fair amount of time.

I believe that these empirical results back up the earlier informal observations. In addition, they pinpoint the relative frequency of ML errors and were part of the motivation for the design of *CYNTHIA*.

2.6 Designing *CYNTHIA*

The previous sections have identified the major problems that novice programmers encounter when learning functional programming languages, in particular ML. This section describes how these results have motivated the design of *CYNTHIA*.

Presently, ML programming environments are insufficient. Typically, they provide the novice user with very little practical help at getting started with ML. There are many difficulties that face the novice ML programmer. *CYNTHIA* is not intended to overcome all of these. It is not a tutoring system intended to replace or augment current teaching methods. Rather, it is a system which provides a degree of interactive support for the user that will eliminate or reduce many of the more trivial programming errors that the novice may come across. These sorts of errors typically bog down the novice user in the early days of programming and hence delay them from getting to grips with the more intricate concepts of the language. It is in this sense and this sense only that *CYNTHIA* can be seen as an aid to teaching. Alternatively, it may be viewed as an environment for programming that could be used not only by novices but by more expert users as well. Admittedly, *CYNTHIA* only supports a subset of the ML language which would deter its use by very experienced users, but the system was designed to support a wide enough subset that is suitable for doing some medium-size programming. This subset could be extended. *CYNTHIA* differs from many programming environments meant for novices in that it does not restrict the user to programming in a particular way. Although it encourages a certain programming style, the user retains a great deal of freedom to edit programs as they wish. This is in contrast to intelligent tutoring systems which restrict the user to a set of pre-stored examples, and editors such as [Bundy *et al.* 91] which are both restricted to a too narrow subset of the language and also place restrictions on the order in which edits should be made. The intention was for *CYNTHIA* to be a realistic, flexible programming environment that reduces the frustration and confusion associated with novices' debugging and hence promotes an increase in the learning curve.

The following features were identified as (partial) solutions to the problems identified in earlier sections:

Programming by Analogy

It was mentioned earlier in this chapter that an important problem-solving technique is to make analogies with previously solved tasks. *CYNTHIA* encourages the use of this technique by being based around the concept of programming by analogy. Rather than writing programs in a text file from scratch, in *CYNTHIA* the user always starts with a previous program and is then provided with a powerful set of editing commands to transform this program. These commands are explained in Chapter 5.

Reduce the Number of Programming Errors

CYNTHIA's main purpose is to reduce the number and frequency of programming errors. This frees the user to direct his thinking to more difficult concepts. *CYNTHIA* analyses programs as they are written. Certain kinds of errors are impossible to make in *CYNTHIA*. Others are possible, but will be made less frequently, and will be clearly signalled when they do occur. Chapter 6 explains the underlying mechanisms for ensuring correctness of *CYNTHIA* programs. There are two points worth mentioning here. First, *CYNTHIA*'s underlying design is based on formal analysis techniques that directly guarantee correctness. Second, the editing commands that the user applies are correct. In most cases, editing commands cannot by definition introduce errors. In some cases, however, the user must pass through incorrect intermediate states and so some commands may introduce errors. However, the user is more aware of these errors both because of *CYNTHIA*'s feedback mechanisms and because he knows the error must have been introduced by the most recent edit.

Recursion and termination

CYNTHIA makes no attempt to explain recursion to the user using any of the techniques elucidated previously. Instead, I recognise the usefulness of eliminating non-terminating programs. Hence, *CYNTHIA* restricts the user to writing terminating definitions. In addition, the system constantly displays a list of the currently *valid* recursive calls. By valid, I mean that if the recursive call was used in the definition, the definition would terminate. This display list is a constant reminder to the user of

the recursive calls available at any given time. Whilst not intended to provide tuition on the use of recursion, it is intended to make the user more aware of the importance of recursion and the concept of termination.

Improved Type Error Feedback

Rather than providing cryptic error messages, *CYNTHIA* highlights a program expression that is responsible for the type error. In general, of course, the *real* source of the error may be in some other part of the program.

Explicit Typing

The user must give a type declaration for each function definition. This is intended to avoid the confusion that seems to be caused by implicit typing and type inference. The type declaration is constantly visible. In practice, the user does not enter a type declaration, but edits the declaration of a previous program. Hence, the burden is much less than forcing the user to start from scratch each time.

More efficient programming

CYNTHIA is equipped with some powerful editing commands that can make complex changes to a definition whilst retaining correctness. In this way, some trivial programming steps are automated.

Pattern Matching

All patterns in *CYNTHIA* are guaranteed to be well-defined. In addition, there is a command to allow the user to incrementally build up non-trivial, well-defined patterns. This command emphasises the concept of pattern matching.

In addition, the system was intended to have the following characteristics:

- To be a usable system. Hence, consideration was given to the interface and run-time performance. Because *CYNTHIA* is based upon a theorem proving system,

the interface had to be designed in such a way that the details of the logic were hidden from the user. Whilst not intended as one of the primary requirements of *CYNTHIA*, the system is based on structure-editing. This was considered the easiest way to incorporate the editing command style. Later versions could be text-based systems where the user could enter text freely whilst applying editing commands intermediately.

- The editing commands (see Chapter 5) were designed to be as compact as possible. The number of commands was kept to a minimum so as not to introduce an additional learning burden. Their function was meant to be as clear as possible and were designed to be at the right level of abstraction so that they could be used to transform between arbitrary programs in such a way that the order of application was not critical.

2.7 Summary

This chapter has examined the most common difficulties faced by novice programmers of functional programming languages. The purpose of the chapter was to convince the reader that current programming environments are insufficient to provide the level of assistance to the user that is required to overcome these problems in an efficient way. Current systems concern the user too much with debugging in the early stages of learning and hence the user is not free to progress to more difficult parts of the language. *CYNTHIA* is intended to overcome such drawbacks by providing an environment in which fewer programming errors will be made. A range of sophisticated techniques have been applied and developed to make this automatic program analysis possible. A major achievement is to package the system in such a way that the user need not be concerned with the theorem proving system that is used in the program analysis.

 'm lost," moaned Mavis to no-one in particular. "I have a backpack full of dodgy maps and guide books plus a compass and all sorts of other stuff. But none of it is any help at all. I even bought a new little tool off some guy on the ship. It's got this display that keeps track of where you've been so you can always find your way back to the store. Trouble is it stopped working as soon as I entered the forest. It was fine on the ship and I have to admit I was very impressed, so I handed over a ridiculous amount of money. But now when it's really needed, it gives up on me. Not that it would get me out of here, anyway. It doesn't tell you what you really want to know. It just tells you where you've been, not where you should be going."

No sooner had Mavis muttered her plea for help than a soft hand touched her on the shoulder. Mavis jumped up, startled. "Who are you?" she asked the young girl that had appeared from nowhere.

"I am Cynthia," came the reply. "Don't worry. Come with me. I'll show you how to make sense of these signs."

"You will?"

"Of course. Just don't let anyone else know I helped you out. This place is supposed to be a secret!" And with that, Cynthia beckoned to Mavis to follow her down the path marked "The correct path".

Chapter 3

Advanced Programming Environments

This chapter presents a survey of programming environments that have been designed with the purpose of providing additional support to the programmer. Often, these systems have concentrated on supporting novices, be they intelligent tutoring systems (ITSs) that lead the student through a series of examples and comment on their solutions, or interactive environments intended to be used in the same way as traditional environments but that provide some limited degree of intelligent feedback or analysis. I first discuss examples of these two types of environment and go on to examine, more closely, approaches that encourage programming by analogy, as defined in Chapter 2. I finish off with a description of systems that provide support for showing the correctness of programs.

3.1 Interactive Programming Environments

This section surveys examples of programming environments intended to give interactive help to the user during the programming process. I usually restrict to systems for declarative languages, but will also mention imperative languages where relevant.

3.1.1 Structure Editors

One of the established ways of programming is to write programs using a text editor and then to pass the text file to a compiler which will check for errors. In contrast, the

basic premise behind structure editors[1] is that programs are not merely text, but are "hierarchical compositions of computational structures that should be edited, executed and debugged in an environment that consistently acknowledges and reinforces this viewpoint." ([Teitelman & Reps 84]). In a typical structure editor, rather than writing text, the user programs with templates or pre-defined program fragments. Templates consist of program constructs with gaps that need to be filled in by the user. For example, a template for a conditional statement in ML might be:

```
if ?Cond then ?Exp1 else ?Exp2
```

where `?Cond`, `?Exp1` and `?Exp2` are as yet unspecified. More complex templates may implement common algorithms. The user selects templates from a given menu which are then inserted into the program at the current point. In addition, program expressions may be edited in a structural way. The user may, for instance, highlight a subexpression of an instantiation of `?Cond` and edit this subexpression in some way. One of the main advantages of structure editors is that the number of syntax errors tend to be fewer. The syntax of the templates is correct and so errors may only occur when instantiating expressions. Although instantiations may also be done structurally, this tends to be long-winded — consider editing the expression `(h::h)::t` to `h::h::t` structurally. Rather than just removing the brackets, the user would have to change `h::h` to `h` and `t` to `h::t`. For this reason, instantiations are usually done by entering text freely and a parser checks the syntax of the entered expressions. Systems such as these that combine structured and text editing are called hybrid editors. The main challenge in designing a hybrid editor is to ensure a smooth transition between text and structured editing. Few systems go on to provide any guarantees of correctness beyond mere syntactic correctness and any further guarantees, if present, tend to be very simple in nature.

Structure editors have been around since the 1970s. One of the earliest was EMILY [Hansen 71] designed for the construction of programs written in higher-level languages. Some ideas from EMILY and also from INTERLISP [Teitelman 75] were incorporated into the Cornell Program Synthesizer [Teitelman & Reps 84], a structure editor originally designed for the language PL/I [Conway & Gries 79] and then PASCAL. Teitelman

[1] often called syntax-directed editors

and Reps make the following claims about the advantages of template-based editors. Templates eliminate mundane programming tasks — there are fewer typographical errors, indentation is automatic, it is productive (short commands can insert long templates). In addition, since templates correspond to abstract units, the process of programming takes place at a higher level of abstraction. The programmer does not get lost in syntactic detail. Of course, many criticisms have been made of stucture editors. Although economical for small programs, they can become frustrating and time-consuming when writing larger programs. In much the same way as editing individual expressions structurally becomes unmanageable, so does editing entire programs if they are sufficiently complex. A good compromise would seem to be to combine the approaches of structure and text editing.

Despite the oft-quoted drawbacks, structure editors are still in use today. Structure editor *generators* are often used to quickly construct prototype editors for new languages. One such generator is the Synthesizer Generator [Reps & Teitelbaum 89], a successor of the Cornell Synthesizer. Although developed in the late 70s, it is still being used for applications [Efremidis *et al.* 93, Liu 95]. The Synthesizer Generator is based upon the concept of attribute grammars [Alblas & Melichar 91]. Attribute grammars are context-free grammars extended by attaching *attributes* to the nonterminal symbols of the grammar and by supplying *attribute equations* to define attribute values. A typical attribute might be the type of an expression. Attribute grammars provide a powerful mechanism for expressing dependencies within a tree and can be used to define editors whereby edits may be propagated throughout a program. For instance, if the user changes a type within an ML program, this change could be propagated so that any expressions within the program that become ill-typed as a result of this change can be notified to the user. More will be said about attribute grammars in Chapter 9.

A more recent environment generator is CENTAUR [Borras *et al.* 88]. The CENTAUR system allows the user to provide a specification of the syntax of a language, as with the Synthesizer Generator, but also the semantics, expressed in a formalism based on Natural Deduction [Kahn 87]. From the syntax specification, CENTAUR generates a structure editor complete with scanner, parser, pretty-printer and abstract syntax tables. From the semantic specification, CENTAUR is able to derive a type-checker,

an interpreter or a translator. As an example, the dynamic semantics of the language EIFFEL were embedded in a CENTAUR implementation [Attali *et al.* 93] and a programming environment generated. CENTAUR was also used as the basis of a JAVA editor [Attali 96] which provides visual tools derived from a structural operational semantics.

3.1.2 Pedagogic Environments

CYNTHIA is not intended only as a novice user environment, but so far its main use has been in a learning context. Hence, it is important to compare it to other learning environments. This section presents the most pertinent research into pedagogic environments. Such environments typically fall into one of two camps. Learning environments are highly interactive environments designed in such a way that the student may experiment with a certain concept, such as a new programming language, but in essentially an undirected way. Typically, students will be provided with teaching material external to the system and will work through this material using the system. In contrast, tutoring systems embody ideas from cognitive psychology and are primarily concerned with taking away from the human teacher some of the burden associated with teaching. Rather than experimenting freely, the student is taken through a series of examples. The tutoring system introduces new concepts and then tests the user's knowledge by analysing students' solutions to set examples. Although learning environments may have quite sophisticated program analysis techniques such as type-checking, tutoring systems will be capable of reasoning about deeper aspects of the programming process, such as good programming style or behavioural correctness. For this to be tractable, tutoring systems are usually restricted to a relatively small range of examples. Students' solutions are often compared to pre-stored expert or ideal solutions and feedback given accordingly. In contrast, learning environments support a much wider subset (possibly the entirety) of the programming language but do not have knowledge about specific examples. *CYNTHIA* falls very much into the learning environment camp.

Many tutoring systems have been developed for programming. One of the most famous is the LISP tutor [Corbett & Anderson 92]. The idea here is that the system is pre-

stored with 240 examples and for each of these has a description of the task and a set of rules for generating solutions to the task taking into consideration various aspects of programming style. These rules are of two kinds. First, there are rules that represent valid steps on the path from task description[2] to solution. These were acquired from studying programmers. Second, rules embody common misconceptions. As the student builds a solution, the system simulates a corresponding expert solution and by comparing the two and also the misconception rules, can provide advice on problems encountered or on what to do next. A rule-based approach is useful in two ways. First, it is easy to generate explanations. If a student becomes stuck, the system can make the next move and generate an explanation of why this is a good thing to do. Second, it means that the number of correct solutions that the system will accept is much greater. A problem of traditional tutoring systems is that they are only capable of recognising minor variations on a small number of pre-stored correct solutions. The LISP tutor does not store correct solutions but generates them instead.

One interesting and controversial point that the authors point out is that of immediate versus delayed feedback. By providing feedback immediately when the system spots a problem, the student is being notified of the problem in the context in which it occurred, not at the end of the programming task, when the context may be unclear. In addition, letting one error pass unnoticed may lead the student to commit further errors and the process of explaining these to the student becomes more and more difficult. Although [Corbett & Anderson 92] shows that students are more successful when receiving immediate feedback, one of the most common complaints from students is that immediate feedback interrupts and frustrates them. It is unclear at present whether students would gain as much if allowed to experiment more freely and learn from their mistakes. This issue is relevant to *CYNTHIA*. The user is never allowed to introduce a syntax error — his input will be rejected until it is syntactically correct. Other kinds of errors, however, can be introduced at will. The user will not be forced to correct them immediately, but the errors will be highlighted. Chapter 8 contains a report on how users of *CYNTHIA* reacted to this.

One novel way of providing feedback is implemented in the Ceilidh [Foubister *et al.* 98]

[2] A task description is an informal description of a problem. Many ITSs which support only a restricted number of examples have a pre-stored description for each example.

system. Ceilidh is a system for organising courseware that was originally used for teaching C at the University of Nottingham but has also been used in ML courses at Heriot-Watt University. Given a well-structured course, typically consisting of tutorials each containing a number of exercises, Ceilidh can be used for the automatic assessment of solutions to these exercises. Students write programs using a template as a guide and at any time they may submit their answer to the system for marking. This mark gives a crude measure of the student's progress. The student may now modify their original solution and re-submit or they may email a human tutor for more in-depth advice. In this way, Ceilidh gives the student freedom to experiment but feedback is available as and when it is needed.

Original versions of Ceilidh based their mark on very low-level, syntactic features of the submitted programs. The ML version [Michaelson 96] includes some semantic analysis. For any mark to be awarded, the program must compile correctly. Hence, Ceilidh offers no help in locating errors found during compilation. For each submitted definition, the student is required to include, as a comment, a type declaration for the definition. This is because the designers believe that the student's ability to provide a type is an indication of their understanding of the function's purpose. The correctness of this type declaration is the first criterion used in the marking scheme. Other criteria include whether the student has correctly used previously defined functions or not as well as a small number of semantic style considerations. The latter are represented in the system as correctness-preserving transformations. Ceilidh attempts to apply as many transformations as possible and the total possible mark is reduced for each transformation that applies. As an example of such a semantic rule, Ceilidh advocates the use of pattern matching wherever possible, rather than a conditional expression with an explicit comparison with a datatype constructor. The rule encoding this is given below:

```
fun name pattern with var =
 if var=constr
 then expression1 with var
 else expression2 with var   ⟹

fun name pattern with constr =
      expression1 with constr |
   name pattern with var = expression2 with var
```

The keyword 'with' is not ML syntax but means that *pattern* contains *var*. The rule removes the conditional statement and replaces *var* in *pattern* by *constr*. As an example,

```
fun length x = if x=nil then 0
                        else 1 + length (tl x);
```

would be transformed using the rule to:

```
fun length nil = 0
|   length x = 1 + length (tl x);
```

Ceilidh has been used extensively in the teaching of ML over a four year period. The conclusion of the research is that while Ceilidh does not improve the academic performance of students, it does remove a large part of the burden of marking from teachers and so frees them for higher-value aspects of teaching.

DrScheme [Findler *et al.* 97] is an example of a learning environment in which "real" programming can be done, but which is also equipped with a number of features to aid learning. Firstly, DrScheme implements a hierarchy of language levels. The motivation is that in the early stages of programming novices may inadvertantly type in syntax that has not been taught yet. This could be confusing to students. To overcome this, the students select a restricted language syntax from the hierarchy and write programs within this restricted set. Selection is, however, left up to the user so that the stronger students are not held back. Secondly, the "read-eval-print" loop (REPL) often causes problems for Scheme novices. Novices may change definitions but forget to make all necessary updates so that the new definition may appear to work, but in fact gives the wrong answer because it is utilising, for example, an auxiliary function that is still present. Expert users learn to overcome this by re-loading their text files which initialises the REPL. DrScheme has a slightly modified REPL which ensures that the REPL is initialised as necessary and so yields a less confusing semantics. One of the problems in Scheme with run-time error reporting is that Scheme expressions, when compiled, are expanded into their canonical forms — for instance, `let` expressions expand into `lambda` expressions. This means that it is difficult to report the location of the error with respect to the *actual* program text. DrScheme overcomes this by keeping track of the macro expansions that the compiler initiates. This allows it to highlight

errors at the exact spot where they occurred in the program text. DrScheme is also equipped with a number of static debugging techniques. Programs checked for syntax can be annotated with arrows connecting bound identifiers to their binding occurrence or binding identifiers to all of their bound occurrences. A more computationally expensive tool is MrSpidey, a static debugger, which analyses the program for potential safety violations using a form of set-based analysis.

The approach of the developers of DrScheme is to provide a realistic programming environment but with certain extras that are more suited to the novice. The designers do not provide menus for introducing templates on the grounds that the novice does not learn by doing this, but will only learn from their mistakes[3]. This goes against the findings of the developers of the LISP tutor, however, and so adds more spice to the cauldron of debate.

3.2 Programming by Analogy

Programming by Analogy is the process of using a previously solved program as a guide for a new programming task. As with all approaches to problem solving by analogy there are three main steps to the analogy:

- **Retrieval** of previously solved tasks from a case base. The retrieval must be based on some *measure of similarity* that captures which aspects of a problem make for good analogies.
- **Selection** of a task from those retrieved. The retrieval stage may provide a number of possible candidates that match the current task. This stage selects one as close as possible to the task at hand.
- **Transformation** of the selected *source* solution to a new solution for the current or *target* problem. The degree of difficulty of transformation depends strongly on the retrieval/selection steps.

A number of systems have been implemented that attempt to help users to program by analogy. They each deal with some or all of the above problems. The particular

[3] Personal communication with Matthias Felleisen.

approach taken by each design determines how much emphasis should be placed on each stage. The main consideration in *CYNTHIA* is with the transformation stage. It is assumed that the choice of source example is easy enough to be carried out by a novice user. Since the user only edits one function definition at a time and since functional programs are generally small and clean, the retrieval/selection steps are simplified considerably. Moreover, the editing commands in *CYNTHIA* are designed in such a way that the choice of a source example is not overly important. Users may make a sub-optimal choice but still get to the target quickly and easily. This is in contrast to some other systems where the starting point is critical so that if the user fails to make the best choice, the transformation stage may have to be abandoned and the first two stages revised.

3.2.1 Retrieval and Selection

A number of improvements to the template-based approach to editors have been developed, variously called programming by plans, cliché programming, programming with skeletons and programming by techniques. Templates merely provide the user with a frame for each construct of the language. In contrast, these other developments provide frames which correspond to particular algorithms or particular programming strategies.

PROUST [Sack & Soloway 92] is a retrieval system which uses programming plans to debug rather than construct programs. Students write a solution using a standard text editor and PROUST then attempts to match the solution to one or more of its pre-stored plans. Based on this matching, it is able to identify and explain bugs. A later version, CHIRON [Sack & Soloway 92], is able to enter into a dialogue with the user concerning the explanations. Plans, in the PROUST sense, are abstract representations of simple algorithms — see Figure 3.1, where the actual plan is in typewriter font and comments are in italics. Because PROUST analyses conceptual errors, it needs to be supplied with a problem specification and subgoals that need to be attempted. This essentially describes possible solutions to the problem and means that PROUST is severely limited to pre-stored examples.

One variable is used in the counter plan; it is called ?Count.
`Variables: ?Count`
A counter plan's template has two parts to it:
(1) the initialization sets the value of ?Count to zero, and
(2) the update increments the value of ?Count by 1 every time the next value is encountered of the sequence of values being counted.

```
Template:
Init: ?Count := 0
Update: ?Count := (?Count + 1)
```

Figure 3.1: A Proust Plan for a Counter.

Linn [Linn 92] addresses the problems of accessing templates when there are a large number of templates available. She presents two template libraries – one for LISP and one for Pascal (with 70 templates each). She suggests the use of hypermedia tools for helping students to store, modify and retrieve templates. She also raises the point that students often find it more difficult to modify old templates than to program from scratch and hence do not deem it worthwhile to look up plans.

There have been a number of works which apply techniques from Case-Based Reasoning (CBR) in systems that support programming by analogy. ELM-PE [Weber 96] is an intelligent tutoring system (ITS) for LISP. Users select source examples from a library which can then be displayed in a window. These examples can merely be used as reminders or they may be copied by cutting and pasting. ELM-PE has no specialist facilities, however, for transforming the retrieved examples.

[Weber 96] describes the need for a method capable of automatically selecting source examples from a library which may include the user's own solutions. The motivation is two-fold. First, it is insufficient for the library of examples to consist only of examples from the course material. In large and complex domains, there will not always be an optimal example from such a library for all tasks learnt during a course. The system stores the student's own solutions as he produces them so that these solutions or *remindings* also form part of the library. In this way, the library is constantly being updated with examples making the selection of an optimal solution more likely. In addition, the retrieval mechanism will be sensitive to the user's own programming style.

Second, the author provides evidence that students are not very capable of choosing the optimal example themselves. In fact, [Weber & Bögelsack 95] show that students tend just to use as source the most recently solved problem without considering whether another example would be more suitable. To overcome this, Weber presents an algorithm for automatically selecting the best example from the library. Evaluation of the algorithm shows that ELM-PE is often more successful at choosing examples than the student.

Similarity within ELM-PE is based upon abstract, semantic representations both of the text book examples and students' solutions. Each time a new (partial) solution is produced, the solution is analysed along with information from a task description, domain knowledge and an individual learner model to produce a tree of concepts and rules used within the program which acts as an explanation structure. *CYNTHIA* includes a representation of programs as proof trees which are similar in some ways to these explanation structures. Both provide a semantic analysis of a program.

ELM-PE uses its explanation trees to retrieve examples. An explanation tree is produced using the task description that then constitutes an expected solution to the problem. A type of organizational similarity is then used to retrieve a library example with a similar explanation structure.

The main problem with ELM-PE is that it is only intended to be used in teaching. Because ELM-PE must be provided with a task description for each problem, the system is limited to a number of pre-defined examples. Although ELM-PE is intended merely as an ITS and not a fully-blown program editor, the author does not address whether his algorithm is able to cope with large libraries. He makes no mention of the library size possible in ELM-PE nor how the usefulness of the algorithm varies with the size.

TED [Bowles & Brna 93, Brna & Good 96, Ormerod & Ball 96] is another system that aims to help students with the task of retrieving a suitable source problem, in this case for Prolog programs. TED is much less sophisticated than ELM-PE. The main difference in the two approaches is that TED makes no attempt to choose between retrieved cases. Based on a set of constraints, specified by the user, which the target

problem must satisfy, all matching source solutions are presented and the user must select between them. Hence, TED deals with only the retrieval stage of the analogy and selection is left to the user. On the one hand, this allows a greater degree of flexibility as the user is not confined to an automatically generated source solution and may view a number of possible candidates. On the other hand, the crucial choice of which source case is best is left to the user who, as shown in [Weber & Bögelsack 95], is fallible.

The constraints that classify solutions are based upon an *intermediate representation language*. To embark upon a new task, the user is asked to specify the number, type and mode[4] of each argument. In addition, the user may specify relationships that hold between arguments, such as "the output argument is an element of the input list" or "the first input argument is compared with an element of the second input argument". These two sets of constraints are then used to retrieve similar source cases.

Note the analogy between these constraints and those used in the constructive logic framework of *CYNTHIA*. Specifying the type and number of arguments corresponds to a weak specification in *CYNTHIA*. Giving relationships between arguments corresponds to stronger specifications. *CYNTHIA* currently uses only weak specifications but a possible future direction would enable a user to specify stronger constraints. Note that the relationships possible in TED are very restricted. The user makes selections from a pull-down menu and hence is restricted to 16 pre-defined possibilities.

A number of other works have also made an attempt at case-based tutoring. The earliest approaches merely provided an environment where the user can browse through a library of examples and display a selected example as a visual aid for the current task. [Neal 89] is a PASCAL editor that allows a user to access a specified example and then edit it by inserting new templates. These systems are very limited, however, as little consideration has been given to how best to transform the retrieved examples.

One interesting idea is to base a retrieval mechanism on the use of types [Runciman & Toyn 91, Rittri 89]. The authors of [Runciman & Toyn 91] claim that an easy-to-use retrieval system would encourage code re-use, where by re-use they

[4] Mode is used here in the Prolog sense — + means that an argument must be instantiated when the procedure is called, - says it should not be and ? says it may be partially instantiated.

mean that previous functional programs are combined to form new definitions without necessarily making modifications to the old function. They use the polymorphic type as a key to retrieve past definitions. [Runciman & Toyn 91] includes an empirical analysis of well-known functional libraries which shows that a search for a component in a library containing 200 definitions is likely to produce about 5 matches. [Rittri 89] requires the user to supply an explicit polymorphic type as a key to use in the retrieval. His idea is then to search the library for components with the same type, but modulo some axioms that state, for example, that the argument order should be ignored and so should the distinction between a curried function and an uncurried one. [Runciman & Toyn 91] attempts to provide a retrieval system that does not require an explicit key type declaration, on the grounds that programmers do not write type declarations in languages like ML. Instead, the claim is that programmers formulate laws early on in the software development phase. If these laws are written in the type discipline, key types can be inferred from them. A law specifying a required function `positions`, returning a list of positions of an element in a list, might be:

```
member p (positions e x) ⇔  item p x = e
```

where `item` takes a position and a list and returns the element at that position. Given the types of `member` and `item`, a type for `positions` may be inferred and used as the search key. If there are several different laws, several keys may be inferred. These can be refined by a process known as co-unification. Neither of the following types is an instance of the other:

```
'a -> int list -> int list list
string -> 'b list -> 'b list list
```

They can be co-unified to give a new key: `'a -> 'b list -> 'b list list`. This process reduces the number of keys and hence reduces the complexity of the retrieval. The final extension to the idea is to look for a higher-order function that, when suitably applied, gives us what we want. For example, re-ordering arguments can be seen as applying the higher-order function `flip`:

```
fun flip f x y = f y x;
```

At present, *CYNTHIA* has no facilities for helping the user to select a source example. Empirical evaluation suggests that such facilities are not necessary and if they were

provided they would only need to be very simple (see Chapter 8). The library examples could be indexed in such a way that the user could browse through them more easily. Possible indices would be: the type of the function, the type of the argument(s) being recursed upon, the kind of recursion. It may be that if *CYNTHIA* was adapted to more advanced users, the importance of a suitable retrieval mechanism would increase.

3.2.2 Transformation

An example of an approach that includes a small element of transformation is [Barker-Plummer 90] which attempts to use plans to allow code re-use. Clichés are commonly occurring generalised (Prolog) procedures. The idea is that the user retrieves such clichés and these can then be instantiated in different ways to produce different concrete procedures. An example cliché is given below. The symbols prefixed by `$` are cliché parameters. `$P/n` is the procedure being defined; `$Q/n` the parameter required to define `$P/n`; `n` is the arity and `&Aux` is an extra argument for the recursion.

```
$P([],&Aux).
$P([H|T],&Aux) :- $Q(H,&Aux),$P(T,&Aux).
```

The cliché would be used by instantiating the parameters, whereby `$Q` could be instantiated differently to produce different concrete definitions. [Barker-Plummer 90] assumes an existing library of pre-defined clichés but one could envisage allowing the user to define his own clichés.

Gegg-Harrison's schemata [Gegg-Harrison 92a] are very similar to clichés. He concentrates on trying to provide a classification of list recursive Prolog procedures and comes up with 14 basic schemata which are then used as the basis of an ITS. The main difference between these schemata and clichés is that the schemata are each designed to capture a particular kind of control flow, whereas clichés are meant to capture common features of procedures, not necessarily based on control flow. In the ITS, students are presented with an example predicate similar to the current task. Solving the task is done by filling in a provided schema.

The main problem with programming plans and schemata is that they tend to be specific to a small class of programs and hence an unmanageable number of them are needed to support a wide range of programming. Gegg-Harrison [Gegg-Harrison 91],

for example, describes 14 schemata but only covers a limited range of recursive programs. This is acceptable in the context of a tutoring system where the student is being taught specific strategies through examples, but it is a major problem in a real programming environment.

Kirschenbaum [Kirschenbaum *et al.* 89] extends the approach by making a separation between *skeletons* which capture the control flow of a procedure and *techniques* which can be used to modify procedures in such a way that the type of control flow is preserved. Hence, skeletons correspond closely to Gegg-Harrison's schemata and techniques carry out operations such as introducing an accumulator, adding an extra argument, etc. Kirschenbaum's skeletons are also not limited to recursive list processing but include also skeletons which are meta-interpreters or parsers. Many of Kirschenbaum's techniques are similar to the editing commands used in the recursion editor [Bundy *et al.* 91] and *CYNTHIA*. Kirschenbaum does not provide an implementation of his ideas but Robertson achieved this to a limited extent in [Robertson 91].

The recursion editor [Bundy *et al.* 91] is an ancestor of *CYNTHIA*. It allows the user to build programs incrementally by applying a sequence of editing commands. However, it was found that the recursion editor was too sensitive to the order in which commands are applied. *CYNTHIA* has an improved design over the recursion editor. See Chapter 9 for an in-depth comparison of the two systems.

Bowles and Brna [Bowles & Brna 93] attempt to overcome some of the problems with the plans-based approach. They introduce their own programming techniques. One of their main aims was to keep the set of techniques small but open to generalisation. Again, however, they are restricted to a relatively small subset of Prolog programs. The techniques are used in the Prolog editor, TED. The techniques capture relationships between an argument position in the head of a recursive clause and the same argument position in the recursive subgoal. For example, the *list head* technique is used to declare that the head value is the list and the subgoal value is the tail:

```
p([H|T]):-
    .....
    p(T),
    ..... .
```

The *list subgoal* technique is the reverse of this:

```
p(T):-
    .....
  p([H|T]),
  ..... .
```

The *before* technique allows an extra subgoal *before* the recursive call:

```
p(X):-
 .....
 g(X,Y),
 p(Y),
 .....
```

There are just five basic techniques which can be combined in a variety of ways (although not all possible ways – see [Bowles & Brna 93]). The paper also suggests how these techniques might be generalized. However, the authors do not give a clear enough picture of what can and what cannot be done using the techniques. A common problem with techniques-based approaches is that the use of the techniques might be counter-intuitive necessitating a learning overhead.

The plans/techniques approach seems very promising. For a wide range of programs to be available, however, typically large numbers of plans are needed. Techniques reduce this number somewhat but it still seems as though a large number of techniques are needed because they are quite specific.

KBEmacs [Waters 85] is a system that combines the cliché approach with that of automatic program generation. The user constructs a program by issuing a series of high-level commands which "can be as much as an order of magnitude shorter than the program it describes" [Waters 85]. The system can be used on ADA or LISP programs but the prototype system proved to be too slow and not robust enough for real practical use. KBEmacs works as follows. Each command executed corresponds to a cliché or algorithmic fragment stored in a library. The commands, written in psuedo-English, also describe how these clichés should be instantiated. KBEmacs then combines the instantiated clichés to produce a program. This approach can yield a significant scale-up – for example, the following series of commands produce an Ada program 56 lines long:

```
> Define a simple_report procedure UNIT_REPAIR_REPORT.
```

```
> Fill the enumerator with a chain_enumeration of UNITS
  and REPAIRS.
> Fill the main_file_key with a query_user_for_key of UNITS.
> Fill the title with ('Report of Repairs on
  Unit ' & UNIT_KEY).
> Remove the summary.
```

These commands contain keywords which refer to standard algorithms known to KBEmacs (e.g. `simple_report`). A particular structure of the data base is also assumed.

This approach, however, is very limited. The authors claim that the commands, written as English expressions, are very similar to an informal description of the current task. However, the commands are still subject to syntactic and grammatical errors and the user must write commands in a certain style. The fact that commands resemble English may lull the user into thinking he can be lax in the command structure. The clichés implemented are also very specific and are restricted to simple tasks. This means that either the user is restricted to a small number of possible programs or that a large number of clichés are needed in which case the user will find it difficult to remember the names of the clichés. The authors themselves admit that thousands of clichés would be needed for a comprehensive understanding of programming by the system. Finally, programs generated by KBEmacs usually require at least a small amount of direct editing by the user. Because the user is unable to follow the development of the program, it will take him considerable time to go through and understand the program so that necessary changes can be made.

3.3 Approaches to Program Correctness

This section looks at systems that deal with program correctness. *CYNTHIA* is an attempt to provide a programming environment that is usable by non-experts but that also applies some techniques from the field of automated reasoning to achieve non-trivial correctness guarantees. As a result, it lies somewhere between the two extreme approaches to program correctness. At one end of the spectrum, syntax-directed editors provide mere syntactic correctness. At the opposite end, program verification systems allow the user to formally specify the behaviour of a program and prove that the program satisfies this specification. The latter are not overly relevant

to this thesis because they require the user to have an expert knowledge of theorem proving. However, it is worth mentioning a couple of systems that are closest to *CYNTHIA*.

Extended ML [Sannella & Tarlecki 86] is a framework for the formal development of correct programs in the Standard ML language. It is a simple extension of Standard ML in which, in addition to code, the user may also express properties of this code in a logic which is a superset of Standard ML. The idea is that the user first writes a high-level specification in Extended ML and then gradually refines this specification by decomposition into smaller specifications or pieces of code. Once this refinement process is complete, the user is left with verified executable code. The approach merges the stages of coding and verification as opposed to the approach where a program is written first and only then is it verified. A large amount of work has been done in formalising the semantics of Extended ML but work is now turning towards providing tools that will enable the user to prove the verification obligations that arise during refinement. [Sannella & Tarlecki 91] claims that in many examples, admittedly small ones, 90% of the obligations are trivial to establish. The developers are currently working on providing theorem proving support within the PVS [Owre *et al.* 92] system. Since Extended ML is an environment for fully specifying and proving program correctness, it is unsuitable as a vehicle for learning programming. A large degree of user effort is required to make specifications and verify them.

One aspect of program correctness that has been studied extensively is that of termination. Very few programming environments provide termination checking — the recursion editor [Bundy *et al.* 91] is the only one of which the author is aware. In contrast, many theorem proving systems rely on termination checkers to guarantee the totality of definitions introduced by the user. A major shortcoming of many of these systems is that only primitive recursive definitions are supported. TFL [Slind 96b, Slind 96a] is a framework for defining total definitions based on ML-style pattern matching. It is meant to be independent of any particular higher-order theorem prover and has been ported successfully to both HOL [Gordon 88] and ISABELLE [Paulson 86]. The approach is based on proving the well-foundedness of definitions. This involves two steps — providing a suitable well-founded relation, R, and second, proving that the recursive

calls of the definition decrease under R. This means that potentially any terminating definition can be written in the system, although it may involve considerable effort on the part of the user to suggest R and to prove that R decreases. Nevertheless, this is acceptable in a theorem proving system where the user is going to be an expert anyway. It is not suitable for a novice programming environment. The alternative is to restrict to a certain class of terminating programs but this typically excludes many useful programs such as the following (from [Slind 96a]):

```
fun variant(x,l) = if member x l then variant(x+1,l) else x;
```

This function increments a variable until it no longer lies in a fixed list and is often used for generating new variable names. No structurally-based termination checker could prove totality of `variant`. TFL, however, allows the user to specify a well-founded relation that would do the job — an example would be to use `(max l) + 1 - x` (as long as $\texttt{x} \leq \texttt{max l}$).

TFL is unusual in that it supports a wide variety of relations. Many approaches assume a *size* measure on datatypes and expend most of their effort in the proofs of measure-reduction [Walther 88a, McAllester & Arkoudas 96]. TFL has yet to support mutual recursion and indeed, most research has avoided the question of mutuality, although some work has been done [Homeier & Martin 98, Giesl 97].

Many other attempts have been made to prove termination. In a theorem proving context, this means providing a suitable well-founded induction scheme. There are an infinite number of possible induction schemes, and choosing a correct one automatically involves major search problems. In traditional theorem proving systems, the choice of induction needed to prove a theorem was made at the start of the theorem proving process [Bundy *et al.* 93, Boyer & Moore 79] but this tends to be insufficient as it does not take into account later stages of the proof. Middle-Out Reasoning [Kraan 94, Protzen 95] is an attempt to delay the choice of induction scheme by the use of meta-variables that only get instantiated later in the proof, once the pertinent proof steps have taken place. However, this work is still in its infancy and current techniques for automating termination in this way are limited.

To reduce the complexity of the problem, [Walther 94] assumed a fixed size measure in proofs of termination. This means that there is no longer any obligation to produce a

well-founded relation as this has been decided *a priori*. [Walther 94] defined the Estimation Calculus within which to prove size measure reduction. The techniques allow a wide range of programs to be shown terminating but can involve considerable theorem proving effort. This requires the use of the automated theorem proving system, INKA [Biundo *et al.* 86]. [Walther 94] does not consider mutual recursion. An approach very similar to this is embodied in NQTHM [Boyer & Moore 79]. When new definitions are made, the system attempts to show termination using a simple size measure. The system only deals automatically with a limited number of cases, however. To rectify this, the user is allowed to provide induction lemmata to the system to enable it to prove termination. Although this allows a wider range of definitions to be shown terminating than [Walther 94], much more user interaction is needed.

Termination has also been studied in the realm of Rewrite Systems. A Rewrite System consists of a set of rewrite rules, which allow a (sub-)expression to be rewritten into a new expression. Termination of a set of rewrite rules has been studied extensively. The general approach is again to provide a well-founded order which decreases for each rewrite rule. A large variety of measures [Dershowitz 85] have been invented for showing termination, one of the most powerful being the recursive path ordering (RPO) which uses terms as well-founded sets. RPOs involve placing an ordering on the function symbols of the language. Given two expressions, E_1 and E_2 with top-level function symbols, f and g respectively, then $E_1 \succ E_2$ if either $f \succ g$ or if $f = g$ and the arguments of f are greater than those of g under a multi-set ordering. A multi-set ordering is just a way to compare sets of expressions such that the set M is greater than M' if some elements of M can be replaced by any number of smaller elements. So, for example, $\{4, 2, 1\}$ is greater than $\{2, 1, 3, 3\}$ because 4 is replaced by $\{3, 3\}$. The key to automating RPOs is the selection of an ordering on the function symbols and an ordering on expressions. The main advantage of RPOs is that they can deal with mutual recursion.

3.4 Summary

This chapter has provided a survey of programming environments that offer more support than traditional approaches using only text editors and compilers. These

environments fall into two classes — those that teach programming by analysing and providing in-depth feedback on student solutions, and those that are intended for realistic programming tasks but that also include techniques for non-standard error-reporting or correctness-checking. Syntax-directed editors merely guarantee correct syntax. Systems using plans and techniques help with correctness by providing a good starting point for the user. Other systems allow the user to fully specify and prove the behaviour of a program, but are currently unsuitable for use in a programming course.

Cynthia was a pretty girl. She had yet to reach full maturity but this only heightened the sparkle in her eye. Her cheeks were colourful and her hair seemed to change colour according to which way you looked at it. She was a clever girl, too. Without her uttering a word, Mavis could tell that Cynthia knew what she was about. After all, she had no problem navigating through this maze. She obviously had some pretty neat tricks up her sleeve.

As Mavis followed Cynthia down the narrow path, she inquired, "Who are you?"

"I've told you. I am Cynthia."

"Yes," said Mavis. "But what do you do? Why are you here?"

Cynthia didn't seem at all reluctant to talk about herself. "I used to be a guide taking people around off the ship — people like you. But it's not easy leading people. Oh, I know the way well enough. But it's pretty difficult trying to explain it to anyone else. You get travellers from all sorts of different places and lots of them don't speak the language. And even if they do speak it, lots of people don't want to listen. They'd rather be off doing their own thing. But then they just get lost."

"I hope I won't get lost," said Mavis, suddenly worried.

"Don't fret. I've made some changes since then. I took some time off and went looking for someone who could help me. Someone who could help me explain things a bit more easily to people like you."

"And did you find anyone?"

"Oh I certainly did. Come, I'll show you." Cynthia stopped at another crossroads. She took the third exit, marked "Danger", and within a few minutes, Mavis noticed a small spiral of smoke coming from somewhere in the distance. As they drew closer, a small cottage came into view. On the door of this cottage, in bold, bright letters was written: "Walter's house".

Chapter 4

Overview of *CYNTHIA*

This chapter gives a description of *CYNTHIA* from the point of view of the programmer. The underlying framework behind *CYNTHIA* is left to Chapters 6 and 7. First, I briefly describe the design of *CYNTHIA*. There follows an extended example to illustrate how programs are written in *CYNTHIA*. Descriptions of the more complex editing commands are given followed by sections on the error feedback facilities.

4.1 The Design of *CYNTHIA*

CYNTHIA is a program editor for the functional programming language ML that supports programming by analogy. Programs are constructed by transforming an existing program from an available library. The user is provided with a collection of editing commands. Each command makes an isolated change to the current definition, such as adding an extra parameter to a function call. The effects of this change are then propagated automatically throughout the rest of the definition. By applying a sequence of editing commands, previously constructed programs can be easily transformed into new ones. In addition, programs produced using *CYNTHIA* are guaranteed free of certain kinds of bugs.

The original motivation behind *CYNTHIA* was to provide a program editor that incrementally checks that programs are correct in some respects. It is natural, therefore, to base the design around established techniques from logic and proof theory which give us a flexible and powerful way of reasoning about the correctness of programs.

CYNTHIA exploits the proofs-as-programs principle, described in Chapter 5. Each program definition in *CYNTHIA* is represented as a proof of a specification in the proof checker *Oyster* [Bundy *et al.* 90]. The specification is a very weak one, describing merely the type of input arguments and the type of the output argument of the definition. *Oyster* implements a variant of a constructive logic known as Martin-Löf Type Theory [Martin-Löf 79]. Each definition in *CYNTHIA* is associated with a corresponding synthesis proof in this logic. The proofs are constructed in such a way that ML function definitions can be easily extracted from them. It is also possible to associate a position within the proof tree to a position within the syntax tree of the program. The synthesis proof guarantees that the corresponding ML program is correct in the following respects:

- **Syntactic correctness.** All proofs must be syntactically correct, hence so must be the corresponding programs.

- **Type correctness.** If the user introduces a type inconsistency into a proof, the proof remains incomplete because the proof obligations corresponding to the type checking of the relevant expression cannot be solved. The corresponding program will contain a type error. The location of the type error can be highlighted to the user by identifying which part of the program the incomplete proof obligations corresponds to.

- **Well-definedness.** A well-defined function definition is one that is neither over- nor under-defined. In terms of ML, this means that all patterns must exhaustively cover the datatype that they are defined over and must contain no overlapping patterns. The following function definition is under-defined:

  ```
  fun addlist (x::xs) (y::ys) = (x:int) + y :: addlist xs ys;
  ```

  Note that there is no pattern for when either of the input lists is empty. Under-defined functions are allowed in ML (they are flagged as warnings at compile-time) but they can lead to run-time errors. In this example, a call to `addlist` will always produce an error.

The following function definition is over-defined[1]:

```
fun length x = 1 + length (tl x)
|   length nil = 0;
```

If `tl` is defined such that `tl nil` gives an exception, then `length nil` will also produce an exception. Swapping the order of the two clauses would work as expected because ML imposes a top-to-bottom ordering on the clauses (essentially ignoring the ambiguity). In general, ML's top-bottom ordering could lead to errors. The user may be unaware that for a certain input the function is defined twice and ML may not pick up the expected value. For this reason, *CYNTHIA* restricts the user to well-defined functions. In `length` above, `x` would have to be replaced by `(h::t)`. Well-definedness is guaranteed in the program because pattern matching in the proof must be well-defined.

- **Static Semantic correctness**. Most static semantic errors cannot appear in the proof. Errors corresponding to undeclared variables or functions can appear and are indicated by incomplete proof obligations similar to the case for type errors.

- **Termination**. All programs in *CYNTHIA* are checked to be terminating. Termination checking is an undecidable problem. *CYNTHIA* restricts the user to a decidable subset of terminating programs. This is the set of Walther Recursive programs [McAllester & Arkoudas 96]. This set contains a wide variety of recursive programs sufficient for use in real programming situations, such as multiple recursions, nested recursions, recursion with accumulators etc.

The design of *CYNTHIA* is depicted in Figure 4.1. Note that editing commands directly affect the synthesis proof and only affect the program indirectly. *CYNTHIA* is equipped with an interface that hides the proof details from the user. As far as the user is aware, he is editing the program directly. In this way, the user requires no knowledge of logic and proof. The user begins with an initial program and a corresponding synthesis proof. These may be incomplete. Editing commands make changes to a particular part of the synthesis proof. This yields a new partial proof which may contain

[1] `tl` returns the tail of a polymorphic list.

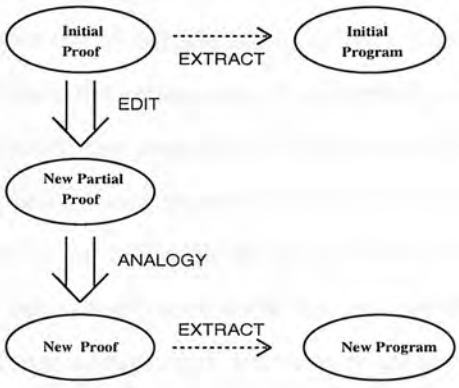


Figure 4.1: Editing Programs in *CYNTHIA*.

gaps or inconsistencies. To fill in these gaps and resolve inconsistencies, an analogical mechanism is used. This mechanism *replays* the proof steps in the original (source) proof to produce a new (target) proof. During this replay, the changes induced by the editing command are propagated throughout the proof. The mechanism attempts to bridge the gaps in the target proof, after which a new program is extracted. This program incorporates the user's edits and is guaranteed correct with respect to the weak specification. Any gaps that could not be bridged are fed back to the user as program errors (e.g. type errors).

In general, constructing proofs by analogy is a difficult task [Melis & Whittle 98]. The specifications in *CYNTHIA* are very weak ones — consisting of just the type of the function plus various lemmas needed for termination analysis. Because we are restricted to specifications involving a limited amount of detail, the proofs are simpler and so the analogy becomes a viable option in a practical, real-time system such as ours. As such, *CYNTHIA* combines proofs-as-programs, analogy and termination analysis in a genuine application.

4.2 Writing Programs in *CYNTHIA*

CYNTHIA is implemented in SICStus Prolog v.3 and Tcl/Tk. Upon startup, the user may select a program from an initial library. The user may edit programs and save them to the library. He will therefore build up his own customised library over time. There are, at present, no facilities for indexing this library in any way.

Figure 4.2 shows *CYNTHIA*'s display. There are three main parts to this. The upper part of the display contains the definition being edited. Note that only one definition is displayed at a time. Other definitions may be selected from the EDIT menu. The user may highlight any part of the program by positioning the mouse over it. Clicking on the left mouse button brings up a menu of editing commands that could be applied at this point. The entries in the menu differ depending on which part of the program has been selected — i.e. only those commands that are currently applicable are given as an option. After selecting a command, the user is presented with a dialog box for him to enter any necessary parameters for the command. He can either enter these parameters as text or select them from a menu of suitable options. When the mouse is moved over a function symbol, the subterm which has that function symbol as top-level operator is highlighted and it is this subterm that is acted upon.

The middle section of the display is used for system messages. The lower part lists all *valid* recursive calls that are currently available for insertion into the program. Valid recursive calls are ones that would not introduce non-termination if used in the definition. *CYNTHIA* uses this list of recursive calls for checking termination of definitions. See §4.4 for further explanation.

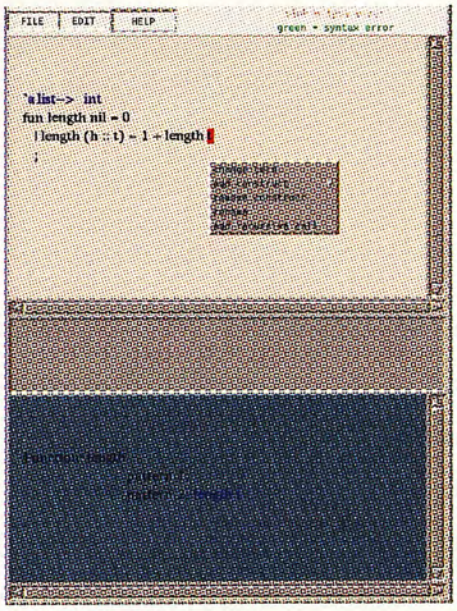


Figure 4.2: Graphical user interface to *CYNTHIA*

CYNTHIA attempts to overcome some of the problems with learning functional pro-

gramming outlined in Chapter 2. Summarising, these are:

- Novices find pattern matching difficult. All patterns in *CYNTHIA* are created using a command for incrementally constructing well-defined patterns.

- ML compilers provide insufficient help with type errors. Users of *CYNTHIA* must provide a type declaration for each function. In practice, because programs are produced by editing old definitions this is not a burden, but merely involves slight modifications of previous type declarations. By forcing the novice to provide a type declaration, the user constantly has the current type of the function displayed and the user becomes more familiar with types. Additional help is provided with types — type errors are precisely located by highlighting them and expressions are checked for type errors as they are entered rather than waiting until compile-time.

- Although no direct help is provided for understanding recursion, all programs must be terminating. *CYNTHIA* constantly displays a list of all recursive calls valid at any given time so the user is fully aware of the recursive calls that may be used. In addition, the fact that the user is making an analogy with the previous successful recursion is helpful.

4.3 An Example

This section gives an example of how *CYNTHIA* might be used to define a collection of functions. A task is presented which might be given to students in the first half of a course on functional programming[2]. The example illustrates a typical situation in functional programming. The student is asked to write a number of table-accessing functions. Each of these functions has a very similar structure. It is natural, therefore, to use *CYNTHIA* as a way of transforming one function definition into another. The example does not illustrate the full capabilities of *CYNTHIA* but gives a typical situation in which it is used. The task is described as follows:

[2] In fact, the task is a slightly modified version of one that is used in the third week of a course on LISP at the University of Edinburgh.

A "table" can be thought of as a store of data-items (the "values") where each entry is indexed by some data-item (the "keys"). The idea is that a value can be retrieved from the table by using the appropriate key, and that there is at most one entry for each key. The assumption is that tables will not be very large, i.e. there is no serious problem concerning the efficiency of search through a table. Choose a suitable representation for such tables and implement the following functions in a consistent manner:

newtable: a new table containing no entries.

addentry k v d: returns a new table which is the same as table d except that the value entered for the key k is v; this value replaces any entry that might be there for k.

findentry k d: given a table d and key k, returns the value entered in d against k. Raises an exception if there is no such entry.

hasentry k d: checks if the table d contains an entry for key k

Let us assume that the student decides to implement the table as a polymorphic list where the odd elements are the keys and the even elements are the values[3]. Let us also assume that the student has already defined `newtable` and now wishes to define `hasentry`. The first thing that the student must do is to decide upon a starting definition. Since the tables will be represented by lists, the student chooses `length`:

```
'a list -> int
fun [ length ]  nil = 0
| length (h::t) = 1 + length t;
```

The first thing to do is to rename this function. If the user selects the command RENAME at the indicated point[4], the definition will change to:

```
'a list -> [ int ]
fun hasentry nil = 0
| hasentry (h::t) = 1 + hasentry t;
```

Note how all of the occurrences of `length`, not just the highlighted one, have been changed.

---

[3] This assumes that the keys and values are of the same type but one could imagine a novice making this sort of assumption. Later, we give an alternative representation that would overcome this problem.

[4] Throughout this thesis, program code enclosed in boxes denotes the point at which the user has applied an editing command.

Next, the student changes the output type of the definition using the command CHANGE TYPE:

```
'a list -> bool
fun hasentry nil = 0
| hasentry (h::[ t ] ) = 1 + hasentry t;
```

The definition is now ill-typed (the underlined expressions are ill-typed with respect to the type declaration). *CYNTHIA* highlights all type inconsistencies in this way. The highlighting serves as a warning to the user but the errors need not be corrected immediately. To access the keys, the program will need to recurse in two steps. To achieve this, the student invokes MAKE PATTERN at the boxed point. This command will replace `t` with two cases – when `t` is empty and non-empty:

```
'a list -> bool
fun [ hasentry ] nil = 0
| hasentry (h::nil) = 1 + hasentry nil
| hasentry (h::h1::t) = 1 + hasentry t;
```

In the third clause, a new variable, `h1`, has been introduced. In addition, a recursive call using this new variable — namely, `hasentry (h1::t)`, has been added to the list of valid calls. It can now be introduced into the program if required. The system knows that any definition involving this new recursive call will terminate. The definition is still missing a parameter for the key to search for. This can be introduced using the command ADD CURRIED ARGUMENT, which adds a parameter *throughout* the definition:

```
'a -> 'a list -> bool
fun hasentry k nil = 0
| hasentry k (h::nil) = 1 + hasentry k t
| hasentry k (h::h1::t) = 1 + hasentry k t;
```

The user gives a name and type for the new argument and the type declaration is updated automatically. Finally, the user needs to change the output in each case.

This can be done using the commands CHANGE TERM and ADD CONSTRUCT(IF THEN ELSE), giving[5]:

```
'a -> 'a list -> bool
fun hasentry k nil = false
| hasentry k (h::nil) = raise excep
| hasentry k (h::h1::t) = if k=h then true else hasentry k t;
```

The above constitutes a reasonable definition of `hasentry`. The student now proceeds to define `findentry` and notices the similarity between the two definitions. To correctly define `findentry`, the user need only invoke RENAME, CHANGE TYPE and CHANGE TERM twice, to give:

```
'a -> 'a list -> 'a
fun findentry k nil = raise excep
| findentry k (h::nil) = raise excep
| findentry k (h::h1::t) = if k=h then h1 else findentry k t;
```

To construct `addentry` is almost as easy. The user needs to invoke ADD CURRIED ARGUMENT, CHANGE TYPE and CHANGE TERM:

```
'a -> 'a -> 'a list -> 'a list
fun addentry k v nil = k::v::nil
| addentry k v (h::nil) = raise excep
| addentry k v (h::h1::t) = if k=h then k::v::t
                            else h::h1::addentry k v t;
```

At this point, the student may decide that he has not chosen the best representation for tables. A better choice would have been a list of pairs. *CYNTHIA* can be used to transform his definitions to suit this new representation. For example, applying REMOVE PATTERN to `hasentry` gives:

```
'a -> [ 'a list ] -> bool
fun hasentry k nil = false
| hasentry k (h::t) = if k=h then true else hasentry k t;
```

Applying CHANGE TYPE at the indicated point gives:

```
'a -> ('a * 'b) list -> bool
fun hasentry k nil = false
| hasentry k ([ h ] ::t) = if k=h then true else hasentry k t;
```

[5] `excep` is a previously defined exception. Because an equality statement `k=h` has been introduced, the second argument should now have type `''a list`. `''a` is an ML *equality* type. It is a polymorphic type over which equality may be defined. *CYNTHIA* does not yet make any distinction between `'a` and `''a`. This is a minor oversight that should be corrected soon.

Applying MAKE PATTERN at the point indicated gives a correct solution:

```
'a -> ('a * 'b) list -> bool
fun hasentry k nil = false
| hasentry k ((h,h1)::t) = if k=h then true else hasentry k t;
```

To evaluate any of these definitions, the user must load them into a compiler. Each function definition may be saved to a file and then loaded by the compiler. At present, each definition is saved in a separate file. This should change in the future.

4.4 Editing Commands

This section gives a description of the editing commands available in *CYNTHIA*. The description is at the level of the user. Chapter 7 describes the underlying techniques. Some commands require that *CYNTHIA* makes a heuristic choice — for example, which parts of the old program should be copied across. Some of these decisions are arbitrary, some have an associated set of heuristics. See Chapter 7 for the latter.

ADD CURRIED ARGUMENT

This command adds a curried argument to a function definition. The user may select an existing input argument and the new argument will be added immediately after it. The type declaration of the definition will be automatically updated. Applying ADD CURRIED ARGUMENT to the following definition.

```
int list -> int
fun sum [ nil ]  = 0
|  sum (x::xs) = x + sum xs;
```

gives

```
int list -> int list -> int
fun sum  nil y = 0
|  sum (x::xs) y = x + sum xs y;
```

where the user has specified that the new argument should be called y and should have type `int list`. Note how all occurrences of sum have been given an extra argument.

ADD ARGUMENT

This command is similar to ADD CURRIED ARGUMENT except that the argument is uncurried — i.e. a variable will be made into a pair. If, in the previous example, the user had chosen ADD ARGUMENT instead, the result would be:

```
int list * int list -> int
fun sum (nil,y) = 0
| sum (x::xs,y) = x + sum (xs,y)
```

The command could be applied any number of times to build up arbitrarily complex pairs of pairs. Note that in Standard ML, a distinction is made between pairs of pairs, such as $(a,(b,c))$ and tuples like (a,b,c). This distinction has not yet been implemented in *CYNTHIA* and so ADD ARGUMENT can only be used to build up the former of these two objects.

ADD CONSTRUCT

This is actually a family of commands that introduce an ML construct at the current point in the program. The ML constructs supported are `if then else`, `case`, `let val`, `let fun`, `fn`. The user will be asked to provide some input such as the name and type of a local function. The constructs introduce branching into the program. If ADD CONSTRUCT is applied to `if then else` or `case`, the original program fragment is copied to all new branches. For example, `if then else` has two branches. If the user applies ADD CONSTRUCT on expression `E` and specifies a condition `C` when asked, then the following code fragment is produced:

```
if C then E else E
```

where `E` has been copied to both of the branches. In the case of `let val` or `let fun`, the original fragment is copied to the body of the let expression, and the declaration is left incomplete. For example, consider the definition:

```
int list -> int list -> int
fun sum nil y  = 0
|  sum (x::xs) y = [ x + sum xs y ] ;
```

If the construct `case` is added and the user supplies an expression `y` as parameter, the result will be:

```
int list -> int list -> int
fun sum nil y  = 0
|   sum (x::xs) y = (case y of nil => x + sum xs y
                           |  (e::es) => x + sum xs y);
```

The pattern given is a default pattern. The user could change this by applying MAKE PATTERN. If `let val` had been introduced at the same point, the result would be:

```
int list -> int list -> int
fun sum nil y = 0
|   sum (x::xs) y = let val z = ???
                        in x + sum xs y
                        end;
```

`???` acts as a place-marker which the user may now edit. Note also that in this case, the user is expected to give the type of the new variable `z`.

ADD RECURSIVE CALL

CYNTHIA is equipped with a termination-checker. The intention is that all definitions written in the system will be terminating. In practice, it is possible to override this restriction in specific cases. *CYNTHIA* maintains a list of *valid* recursive calls which are available for insertion into the current definition. The user may add to this list by using the command ADD RECURSIVE CALL. *CYNTHIA* will only allow a recursive call to be added if its use in the definition would preserve termination of the function. Note also that recursive calls must be applied to all of their arguments (i.e. partial evaluation of recursive calls is not allowed). The list of recursive calls is displayed to the user. Consider the following definition of `ssort`:

```
fun ssort nil = nil
|   ssort (x::xs) = let val z = min (x::xs)
                       in z :: ssort(replace(z,x,xs))
                       end;
```

The function `replace` has previously been defined. It has input type `int * int * int list` and replaces all occurrences of its first argument in the list by the second argument. When `replace` was defined, *CYNTHIA* automatically derived a bounding lemma for `replace` — namely, that the length of `replace(x,y,z)` is no bigger than the length of `z`. *CYNTHIA* can now use this lemma to verify that the recursive call `ssort(replace(z,x,xs))` is valid. The idea is that |`replace(z,x,xs)`| is no bigger

than |xs| and is hence strictly less than |x :: xs|. Therefore, `ssort` reduces the size of its argument and so will terminate. This is Walther Recursion — see Chapter 5. If the user attempts to introduce a non size-decreasing argument, *CYNTHIA* will produce an error, indicating that *CYNTHIA* cannot guarantee termination if the recursive call is used. Hence, recursion in *CYNTHIA* is a two-step process — first, a recursive call is declared using the command ADD RECURSIVE CALL and second, the recursive call is entered into the definition.

MAKE PATTERN

As previously mentioned, a common technique when writing ML programs is to base definitions on pattern matching. MAKE PATTERN is a command that allows the user to incrementally construct arbitrarily complex patterns. It can be applied to define the top-level function, to define a locally declared function or at a `case` node. The patterns produced in this way are guaranteed to be well-defined. MAKE PATTERN is similar to the *create pattern* command in the proof editor ALF [Coquand 92]. The user can highlight a variable of a certain type. The application of the MAKE PATTERN command splits the object into a number of patterns — one for each constructor function used to define the type. Hence, MAKE PATTERN on `x`, of type list, below

```
fun f(x ,l)=..
```

produces two patterns:

```
fun f(nil,l)=..
|   f(h::t ,l)=..
```

Two new variables `h` and `t` have been introduced. More complex patterns can be defined by applying the command a number of times. Highlighting `t` and applying MAKE PATTERN again gives:

```
fun f(nil,l)=..
|   f(h::nil,l )=..
|   f(h::h2::t1,l)=..
```

This can be done for other datatypes by using the definition of the type as encoded in ML. Suppose `l` is of type `'a tree` where `'a tree` is defined as:

```
datatype 'a tree = leaf of 'a | node of 'a * 'a tree * 'a tree;
```

then the indicated occurrence of `l` can be split to give:

```
fun f(nil,l)=..
|   f(h::nil,leaf x)=..
|   f(h::nil,node(x,l1,l2))=..
|   f(h::h2::t1,l)=..
```

MAKE PATTERN cannot be applied to datatypes such as `int` or `string` which do not have a finite number of constructors (see Chapter 7). Neither can MAKE PATTERN be applied to purely polymorphic types such as `'a` because there are no constructors in this case[6]. Note that MAKE PATTERN always produces well-defined patterns, even when nested patterns are created.

CHANGE TYPE

As previously stated, the user must give a type declaration for every function defined in *CYNTHIA*. During the course of a program the user may realise he has stated the top-level type incorrectly. Or he may want to change the top-level type of an old program to produce a new one. *CYNTHIA* provides quite advanced facilities for doing this. Define a parameter to be *split* if the parameter is defined by pattern matching (i.e. MAKE PATTERN has been applied to the parameter one or more times). There are two cases to consider when changing the type of an argument in the type declaration. First, if the argument is split, *CYNTHIA* will automatically update the patterns caused by the split. This involves an underlying structural change to the definition. If the argument is not split or if it is the result type that is being changed, no such manipulation is necessary. Note that type inconsistencies could, however, be introduced into the definition. §4.5.1 explains how these are dealt with.

Consider the case when the parameter is split. The following is a definition of `length`:

```
 'a list  -> int
fun length nil = 0
|   length (h::xs) = 1 + length xs;
```

Suppose the user wishes to change the type of the first argument from `'a list` to `'a tree`. He can click at the indicated point and select the CHANGE TYPE command. This edits `length` into:

[6] It can, of course, be applied to polymorphic types like `'a list`.

```
'a tree -> int
fun length (leaf n) = 0
|   length (node(h,xs,ys)) = 1 + length xs;
```

In this case, *CYNTHIA* will also add the recursive call `length ys` to the list of valid recursive calls. This is because any definition using `length ys` in the second clause is guaranteed to be terminating. More generally, *CYNTHIA* will add recursive calls for the new type that do not compromise termination. More complicated examples arise when MAKE PATTERN has been applied more than once. If the original program had been:

```
fun app2 nil l2 = l2
|   app2 (x::nil) l2 = (x::l2)
|   app2 (x1::x2::xs) l2 = x1::(app2 xs l2);
```

there is a choice as to what the program should look like after a change of type of the first argument to `'a tree`. *CYNTHIA* looks for a mapping between the old and new datatype definitions and uses heuristics to select a mapping if necessary. This mapping is then applied to the old pattern definition to produce a new one. The result in this case is:

```
fun app2 (leaf n) l2 = l2
|   app2 (node(x,leaf n,t1)) l2 = (x::l2)
|   app2 (node(x1,node(x2,xs,t2),t3)) l2 = x1::(app2 xs l2);
```

where `n`, `t1`, `t2` and `t3` are fresh variables. A full description of this technique is provided in Chapter 7.

CHANGE TERM

This command allows text to be entered freely on the RHS of an equality. By clicking on an expression, E, and selecting CHANGE TERM, the user may edit E using most normal text-editing facilities. Note, however, that any new program constructs such as `if then else` must be added using the command ADD CONSTRUCT.

Other Commands

In addition to the commands described so far, there are a number of lower-level commands for making minor changes to definitions. RENAME globally renames a variable,

CHANGE TERM allows free-text editing of an expression and UNDO removes the most-recent edit. In addition, there are commands for removing parts of the definition — REMOVE ARGUMENT, REMOVE CURRIED ARGUMENT, REMOVE CONSTRUCT and REMOVE PATTERN. For a fuller description of these editing commands, see Appendix B.

The editing commands were designed with the intention that any definition may be transformed to any other definition[7]. Of course, it makes sense to choose as a starting function a definition that is close to the target function. However, the user will not be overly disadvantaged by making a sub-optimal choice. The commands fit together in such a way that it is easy to recover from an incorrect application of an editing command, even if other edits have been applied since. The intention also was to keep the set of commands as small as possible. This means that the commands are easy to learn and that very little experience of *CYNTHIA* is needed before one can start editing programs. A wider range of commands could have been included, but it is thought that these would have added confusion to the system whilst only providing limited increases in functionality.

4.5 Correctness of Programs in *CYNTHIA*

One of the main motivations behind *CYNTHIA* was to provide an environment for writing functional programs that involved incremental correctness checking. Any definition written using *CYNTHIA* is guaranteed to be correct in the following senses.

- Correct syntax.
- Well-typed.
- Correct static semantics.
- Terminating.
- Well-defined patterns.

These concepts were defined at the beginning of this chapter.

[7] within the subset of ML supported

4.5.1 Incorrect programs in *CYNTHIA*

Although the initial idea was that at any one time the current definition in *CYNTHIA* would be correct, this is not possible in practice. When transforming one definition to another, it may be necessary to go through intermediate stages where correctness does not hold. As an example, consider the following definition:

```
'a list -> 'a list
fun doublist nil = nil
|   doublist (h::t) = 2*h :: doublist t;
```

Suppose the user wishes to modify this into a length function. This involves a change of type of the output of the function. In particular, we will need to change the output in the first clause to `0`. This is now an ill-typed definition — there are two clauses, one having a result type `int` and one `'a list`. There is clearly no way, while having a compact collection of general purpose editing commands, to retain well-typed programs at all points in the editing process. There follows a description of how this problem is overcome in *CYNTHIA*.

Syntactic correctness

Expressions are checked for syntactic correctness at the time at which they are entered and those that cannot be parsed are rejected. Typically, the user would select a command such as CHANGE TERM and a dialog box will appear in which they can enter the expression as they would in a text editor. The expression will only be accepted if it is syntactically valid. This mechanism allows the user to temporarily escape from the demands of the structure editor and to enter text freely. Note that syntax errors can occur when the user is entering text freely but because all incorrect input is rejected, incorrect syntax is *never* introduced into the program itself.

Static Semantic correctness

As with the previous case, all text that is entered freely is checked for static semantic errors. This means that no static semantic errors can be introduced directly by the user. However, incorrect semantics can be introduced indirectly. This will generally happen

if the user applies an editing command that removes an object from the definition. Suppose the user has the following program:

```
fun add x [ y ] = x + y;
```

Applying REMOVE ARGUMENT at the point indicated produces:

```
fun add x = x + y;
```

Note that the right hand side of the equality now contains reference to the variable y which is not declared. Hence, *CYNTHIA* contains an incorrect definition. There are two possible ways to deal with this. One would be to delete the expression y perhaps replacing it with some marker representing a gap. However, it may be that the user wishes to reintroduce y in some other way — for example, as a locally declared variable — in which case, the user would have to edit the right hand side of the equality to re-insert y. The approach followed in *CYNTHIA* is to leave y in place but to highlight it in a different colour (green) so that the user is aware of the syntax error. However, there is no obligation on the user to immediately rectify this error. This leaves him with the freedom to make other edits and come back to correct the error later. If he were to re-introduce a declaration for y, the highlighting would automatically disappear.

This kind of error can also arise when a definition is loaded from file and contains references to a function that has not yet been defined. For instance, the user may save two function definitions to separate files but then re-load them in the wrong order.

Correct Typing

Type errors can be introduced in two ways — either directly via entering text freely or indirectly through the editing commands. The user is allowed to introduce type errors because, as explained previously, intermediate stages must be allowed where correctness does not hold. To help the user to keep track of type errors, type errors are highlighted in pink. In general, it is very difficult to highlight the *actual* source of a type error [Duggan & Bent 96]. This is because the type inference algorithm breaks down when a type is derived that is inconsistent with a previously derived type. However, expressions early on may have a type derived that is correct in the local context but incorrect when the program is considered as a whole. This is not detected until later

on. Hence, type errors may be reported at locations distant from their actual source. CYNTHIA is less susceptible to this problem because each definition must be given a type declaration. The highlighting is then made with respect to this definition. The other reason why CYNTHIA can do better is because large fragments of code are guaranteed well-typed *a priori* because they are introduced by editing commands.

On its own, the highlighting mechanism is generally not enough to track down type errors in difficult cases. CYNTHIA is equipped with a couple of other features that provide further support. First, the user may click on any expression and the type of that expression will be output to the screen. In the case of a type error, the *actual* type of the expression will be output, along with the *expected* type. The expected type is calculated from the top-level type and the types of terms within the vicinity of the expression. This mechanism tells the user why the type-checking has broken down. Second, CYNTHIA is able to explain its type algorithm. Upon request, the user is provided with a chain of reasoning explaining how the system arrived at the expected type. For example, in the following definition:

```
'a list -> 'a list
fun reverse nil = nil
|   reverse (h::t) = reverse t @ h;
```

the explanation provided would be:

```
@ has type 'b list * 'b list -> 'b list.
  (reverse t) has type 'a list. It should have type 'b list.
    Unifying 'b list and 'a list.
  h has type 'a. It should have type 'a list.
    Unifying 'a list and 'a.
    Failure to unify.
```

Type explanation facilities do exist for the full ML language (including type inference) [Duggan & Bent 96]. However, the explanations produced tend to be lengthy and difficult to understand. Note that the explanations produced in CYNTHIA will be much shorter because only one expression, not a whole definition (or file of definitions), is being explained at once, and because the top-level type has to be specified in CYNTHIA.

CYNTHIA's type explanation facility is as yet immature and a thorough evaluation has not been undertaken.

Well-definedness

It is usually impossible to create definitions that are ill-defined. All patterns are produced using the MAKE PATTERN command and MAKE PATTERN always produces well-defined patterns. The only exception to this is patterns on integers (see below).

Termination

Most programs in *CYNTHIA* are guaranteed to be terminating. Generally, whenever a new recursive call is introduced the definition is checked to see if it terminates. If not, the user is informed (by a simple system message) and the recursive call is rejected. Checking termination is an undecidable problem. Hence, there are terminating programs that cannot be defined using *CYNTHIA*. The termination checker is modular, however, and so it would be easy to modify *CYNTHIA* so that the termination restriction could be over-ridden.

In some cases, non-terminating programs are accepted. This is possible in the current version in the particular case of recursion on integers. The following definition is non-terminating on negative arguments:

```
fun t 0 = 0
|   t n = t (n-1);
```

However, it is a very natural definition and will terminate for non-negative integers. *CYNTHIA* provides this function in its library. The user is free to modify the definition as he wishes. In effect, *CYNTHIA* is allowing a non-terminating definition but under a proviso — that the function is not called with a negative parameter[8]. The alternative would have been to force the user to write a terminating definition (using a conditional statement that gives a value for `n <= 0`) but the above definition is more natural. Dealing with integers is generally problematic within *CYNTHIA*'s framework. The above solution, of allowing users to edit ill-defined definitions using integer recursion,

[8] though *CYNTHIA* does not enforce this

was meant to be a temporary one. A more principle approach was implemented but not in time for use during the evaluations. This approach is presented in Chapter 7.

4.6 Novices and *CYNTHIA*

Chapter 2 describes experiments with novice learners of ML at Napier University and identifies some shortcomings of current learning processes. The results of these experiments directly affected the design of *CYNTHIA*.

The editing commands in *CYNTHIA* can be categorised into structural commands that change the overall program structure and non-structural commands that make only local changes. An example of a structural command would be CHANGE TYPE when a pattern transformation is involved. Non-structural commands are those such as RENAME or CHANGE TERM. The intention behind this division was to encourage (although not enforce) a degree of top-down programming. This style of programming is extremely useful when first learning functional languages, especially if the student's background is in procedural languages. The ideal way to use *CYNTHIA* is to use the structural commands to construct the correct program structure and then to use the non-structural commands to fill in the details. This is in many ways very similar to the approaches of using templates or schemata [Kirschenbaum *et al.* 89, Gegg-Harrison 92b].

One of the major problems with current ML environments is that it is very difficult for students to locate and correct errors. As a result, students spend a large proportion of their time debugging often trivial errors. As mentioned in Chapter 2, trivial syntax errors that would be easily spotted by experts are not spotted by novices because they lack the understanding of the relevant concept and so do not realise that an error is present. When designing *CYNTHIA*, two ways were devised to overcome this. First, *CYNTHIA* restricts users, as much as possible, to correct programs. Hence, they will encounter fewer errors. Second, once an error has been encountered, it should be easier to locate because of the incremental nature of the editor. If the user applies a command to a program fragment, which results in an error (flagged by the system), the student knows that his last edit must have introduced that error. Hence, errors are trapped as the program is written rather than waiting until compile time.

More specifically, novices often have difficulty deciding upon the correct pattern definition for a function. They are capable in simple cases where the function only has one argument that is pattern matched against, but become lost when more than one argument is pattern matched or when there are nested patterns. The MAKE PATTERN command overcomes this problem because it allows the user to build up patterns that are automatically well-defined.

One of the major difficulties that novices have is with typing. Novices can ignore types and be unaware of type inconsistencies which arise. Compilers generally provide error messages that are unhelpful and merely cause confusion. In *CYNTHIA*, the user must declare the top-level type of the function being defined. This forces the novice to think about the types and hence reduces the type errors in the rest of the program. Once the top-level type has been given, the types of expressions in most other parts of the program are determined and hence need not be given. The other advantage of having a top-level declaration is that the type error messages are more likely to be focussed on the user's wishes. All type feedback is given with respect to the top-level declaration, so if the user knows this to be correct, it should be easier to figure out why type errors have occurred.

Recursion is well-known to be a difficult concept to learn [Anderson *et al.* 88]. Novices can have considerable difficulty with even primitive recursion schemes. An introductory course will also introduce non-standard schemes involving accumulators, multiple recursion, course-of-values recursion and nested recursion. To help novices keep track of what happens within non-standard recursions, they are forced to enter the recursive calls via the command ADD RECURSIVE CALL. The valid recursive calls are constantly displayed in a special window. The idea is that this will make the student more aware of the importance of recursion and more aware of the concept of termination.

Chapter 8 addresses to what extent *CYNTHIA* has been successful at overcoming such problems with learning ML.

4.7 The Scope of *CYNTHIA*

CYNTHIA supports a purely functional subset of the core SML language. It is meant

to be an environment for editing function definitions and as such does not support evaluation of expressions. Currently, datatype declarations are not included. Datatype declarations are a way of creating user-defined types. An example (`'a tree`) was given earlier in this chapter. References and record types have also been omitted. Appendix D gives a grammar of the subset of SML which has been implemented in *CYNTHIA*. The scope turns out to be sufficient for a typical first course in ML. The main omissions of the core language are that mutually recursive functions cannot be defined and user-defined types are not supported. There are problems dealing with integers and tuples. At present, a number of datatypes that would normally be user-defined are built-in — examples are trees and queues[9]. The main problem with introducing mutual recursion is that our termination checker does not deal with mutual recursion. However, *CYNTHIA* could be extended with another termination checker. One could imagine having a number of termination checkers available and control could be switched between them as necessary. Indeed, the termination check could be switched off altogether if the need arises.

4.7.1 Datatypes Supported

I give here a description of which datatypes are supported by *CYNTHIA*. In general, any ML datatype may be used if built up from the standard type constructors. This includes function types, pair types, polymorphic types, list types etc. The current implementation does not allow user-defined datatypes but this is an implementation failing only. Minor extensions would allow the user to define any ML user-defined type. There is a restriction which limits the datatypes that can be used, however. These arise because of the restriction to terminating, well-defined function definitions.

Consider the following definition.

Definition 1 *Suppose a data constructor c has type* $T_1 * \ldots * T_n \rightarrow T_0$. *Then c is a base constructor for* T_0 *if* T_0 *is distinct from each* T_i, $1 \leq i \leq n$. *Otherwise, c is a step constructor for* T_0.

[9] The CHANGE TYPE command is not restricted to built-in types but would work on user-defined types if they were available.

The restriction is as follows. Suppose a function is recursive and is defined via pattern matching. In this case, the definition will look something like:

```
fun f ... (b_1 ( ... )) ...  = ...
⋮
|  f ... (b_k ( ... )) ...  = ...
|  f ... (c_1 (x_11,...,x_1n1)) ...  = ...  (f ...  x_11j ... )
⋮
|  f ... (c_l (x_l1,...,x_lnl)) ...  = ...  (f ...  x_llj ... )
```

where each b, c are base, step constructors respectively. For this kind of definition to be accepted by *CYNTHIA*, each base constructor clause must contain no recursive calls on the RHS of the equality. In addition, the type of each x_{rr_j} must be the same as the type of $c(x_{r1}, \ldots, x_{rn_r})$.

This restriction is imposed by Walther Recursion. It essentially says that recursion must be over a measure decreasing argument of a step constructor. It outlaws definitions such as the following:

```
datatype term = Var of string | Atom of string | Term of string * term list;

term -> int
fun count (Var str) = 1
|   count (Atom str) = 1
|   count (Term(str,l)) = sum (map count l);
```

because `Term` is not a step constructor but recursion is over an argument of `Term`. See §7.2.5, p. 166 for a deeper discussion. For a complete description of the restrictions imposed by Walther Recursion, see §6.1.1, p. 114.

4.8 Summary

This chapter has discussed some of the design issues behind *CYNTHIA*. The discussion has been used to motivate the collection of *CYNTHIA*'s commands. Each command has been presented and it has been explained how commands can be combined to construct ML programs.

"This is Walter" introduced Cynthia.

"Hello," said Mavis. She walked towards Walter to shake his hand, just as she had been taught in finishing school. But no sooner had she gone a few steps when Walter suddenly cried out "STOPPPPPPP!" The sound was a blood-curdling, high-pitched screech that stopped Mavis in her tracks. She was beginning to think that maybe coming here wasn't such a good idea after all when Cynthia interrupted. "Walter, you'll scare the poor girl."

"So sorry," whispered Walter. He continued very quietly, "This place is full of holes. Don't ask me why anyone would want to build a cottage in such a place. But no-one ever told be about the holes when I bought it. That hole there is fatal. One more step and it would have been – well, let's just say it's a long way down to the bottom. In fact, I'm not even sure there is a bottom."

"I suppose I should thank you," said Mavis shyly.

"Think nothing of it, my girl! That's what I'm here for. Sit down. Please." Mavis went to sit on the nearest chair but just before her rump hit the seat, Walter was screaming again "NOOOOOOO! NOT THEEEEERE!"

"What's wrong?" asked Mavis. "I can't see any holes."

"No, its not holes, my dear. I'm afraid that chair has been bewitched. As soon as anybody sits on it, it lifts itself off the ground and goes flying about the room like a pecky-feed-bat. It takes hours to get it down."

"Who would want to bewitch a chair?" asked Mavis, perplexed. "And what's a pecky-feed-bat?"

"Oh, plenty would bewitch a chair round here," warned Walter. "Not least those dastardly wolves that lurk about. You be careful of them. Once they get their teeth into something, they never let go. Not ever."

"Walter, do calm down," said Cynthia. And to Mavis, "He gets terribly excited sometimes. Come, we should be getting on. See you later, Walter."

Cynthia led Mavis outside again. Walking back up the path, Mavis noticed that the sign they had followed no longer said "Danger" but had changed to "No danger here". "This really is a strange place," she thought to herself.

Chapter 5

Proofs-as-Programs

5.1 Introduction

CYNTHIA automatically analyses the user's ML programs as they are edited. This chapter presents the framework within which *CYNTHIA*'s program analysis techniques reside. The central idea is to exploit the *proofs-as-programs* idea which defines a correspondence between functional programs and constructive logic proofs. A key aspect of the analysis is to show termination of the programs edited in *CYNTHIA*. This chapter explains the proofs-as-programs idea.

5.2 Proofs-as-Programs

The main idea behind *CYNTHIA* is to exploit the duality between functional programs and proofs in a constructive logic as a framework for the automatic, incremental analysis of ML programs. This section describes the notion of proofs-as-programs, a catchphrase that has come to signify this relationship.

[Howard 80] identifies that in a constructive logic, there is an isomorphism between inference rules and functional terms of the λ-calculus. The λ-calculus is "based on a function-notation principally designed for denoting higher-order functions by Alonzo Church in the 1930s." [Hindley & Seldin 86]. The λ-calculus can be viewed as a functional programming language, so there is a one-one correspondence between functional programs and constructive logic proofs.

A constructive logic is one in which so-called 'pure existence' proofs are disallowed. Consider the conjecture that there exists an infinite number of prime numbers. A classical proof might be based on assuming the converse and deriving a contradiction. This is forbidden in constructive logics — the proof must instead compute a *proof object*. The proof object is essentially a program that incrementally computes the infinite number of primes (or computes another one greater than the ones known so far). Intuitively, the proof object is a procedure for constructing a *witness* for an existential variable. This is exactly what programs do.

One particular constructive logic that has attracted a great deal of attention is that known as Martin-Löf Type Theory [Martin-Löf 79]. Within type theory, each mathematical sentence is considered as a type, the elements of which are proofs of that sentence. A mathematical sentence is assumed to be true *if and only if* there is a proof of that sentence — i.e. the type is inhabited. The type system in Martin-Löf is a similar, but more general version of the ML type system. Types are built up from a selection of base types — such as integers, natural numbers, the empty type — and more complex types are constructed from these. All the base types and those types constructed from them inhabit the type $u(1)$ which is the first of a cumulative hierarchy of universes, $u(j)$ where if $i < j$, then membership and equality in $u(i)$ are just restrictions of membership and equality in $u(j)$. $u(j)$ is closed under all the type forming operations except formation of $u(i)$ for $i \geq j$. Equality in $u(j)$ is the restriction of type equality to members of $u(j)$. This hierarchy guarantees that universes cannot contain themselves, thus avoiding Russell's paradox. The notation $obj : type$ is used to mean that obj inhabits the type $type$.

Because of the fact that mathematical sentences are represented as types, the statement $t : T$ can in fact be read in three ways:

- t is an element of type T
- t is a proof of T
- t is a program for the task specified by T

The next section gives a flavour of the duality between proofs and programs in Martin-

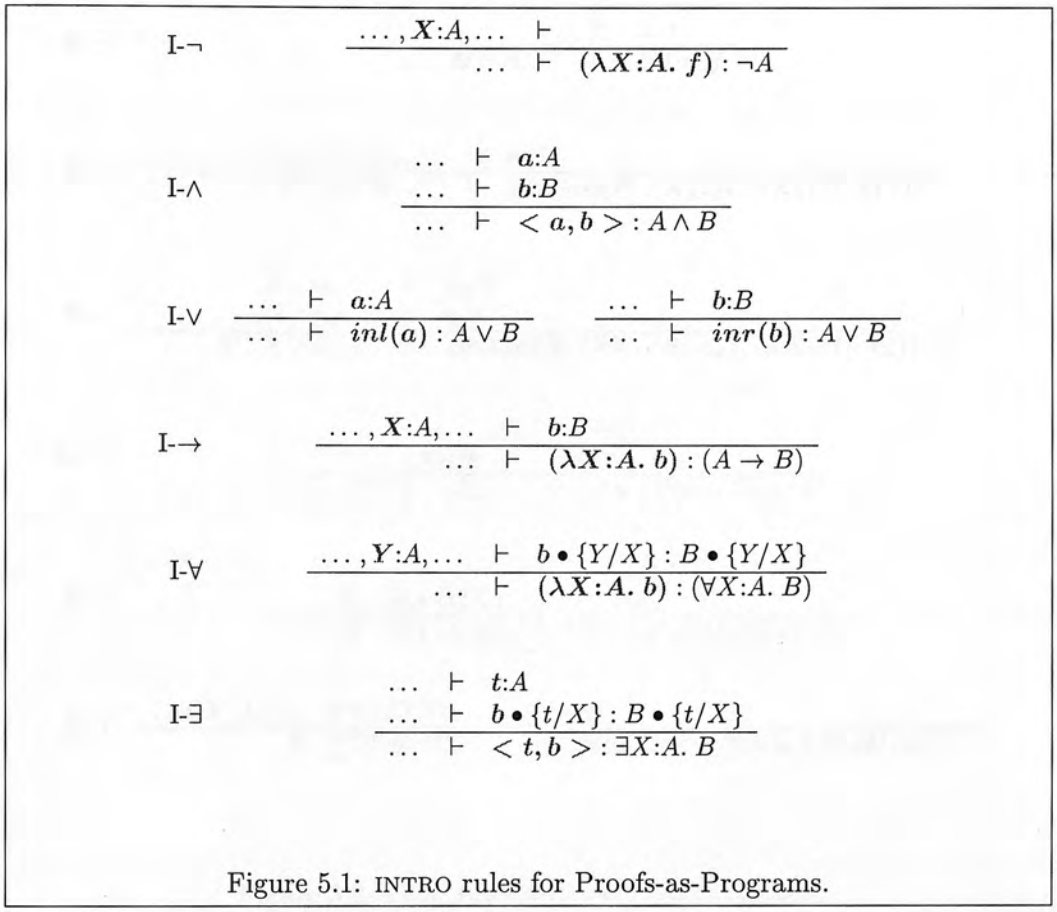


Figure 5.1: INTRO rules for Proofs-as-Programs.

Löf Type Theory, but the reader is referred to [Martin-Löf 79] for details.

5.2.1 Duality Between Rules and Program Fragments

This section gives some of the proof rules associated with proofs-as-programs. Rules are presented in sequent calculus form:

$$\frac{H_1 \vdash G_1 \quad \ldots\ldots \quad H_n \vdash G_n}{H \vdash G}$$

The H_is are hypotheses and G_is are goals. The rule states that if each G_i can be proved assuming H_i, then G can be proved assuming H. The main set of rules is given in Figures 5.1 and 5.2. Explanations, where necessary, are given in the following text.

The inference rules fall into the following categories.

$$\text{E-}\neg \quad \frac{\ldots \vdash a{:}A}{\ldots, H{:}\neg A, \ldots \vdash H(a){:}f}$$

$$\text{E-}\wedge \quad \frac{\ldots, X_a{:}A, X_b{:}B, \ldots \vdash c{:}C}{\ldots, P : A \wedge B, \ldots \vdash spread(P, \lambda X_a{:}A.\ \lambda X_b{:}B.\ c) : C}$$

$$\text{E-}\vee \quad \frac{\begin{array}{l}\ldots, X_a{:}A, \ldots \vdash c_a{:}C \\ \ldots, X_b{:}B, \ldots \vdash c_b{:}C\end{array}}{\ldots, X : A \vee B, \ldots \vdash decide(X, (\lambda X_a{:}A.\ c_a), (\lambda X_b{:}B.\ c_b)) : C}$$

$$\text{E-}\rightarrow \quad \frac{\begin{array}{r}\ldots \vdash a{:}A \\ \ldots, X_b{:}B, \ldots \vdash c{:}C\end{array}}{\ldots, F : A \rightarrow B, \ldots \vdash c \bullet \{F(a)/X_b\} : C}$$

$$\text{E-}\forall \quad \frac{\begin{array}{r}\ldots \vdash t{:}A \\ \ldots, X_b : B \bullet \{t/X\}, \ldots \vdash c{:}C\end{array}}{\ldots, F : (\forall X{:}A.\ B), \ldots \vdash c \bullet \{F(t)/X_b\} : C}$$

$$\text{E-}\exists \quad \frac{\ldots, Y{:}A, X_b : B \bullet \{Y/X\}, \ldots \vdash c{:}C}{\ldots, P : (\exists X{:}A.\ B), \ldots \vdash spread(P, \lambda X{:}A.\ \lambda X_b{:}B.\ c) : C}$$

Figure 5.2: ELIM rules for Proofs-as-Programs.

INTRO **rules**

These rules manipulate a goal G_i on the right-hand side of the sequent, e.g. to introduce an existential witness. An INTRO rule for logical conjunction is as follows:

$$\frac{\ldots \vdash a : A \qquad \ldots \vdash b : B}{\ldots \vdash < a, b >: A \wedge B} \text{ I-}\wedge$$

This can be interpreted as follows. If a is a proof of A and b is a proof of B, then $< a, b >$ is a proof of $A \wedge B$. Alternatively, it can be interpreted in a programming sense — if a is a program which computes A and if b computes B, then the sequence $< a, b >$ (i.e. do a first then b) computes both A and B. Each connective of the logic also has an INTRO rule. For example, look at the rule I-$\vee$ in Figure 5.1. This rule says that if a is a proof of A then $inl(a)$ is a proof of $A \vee B$, where inl is an indicator as to which disjunct has a proof. Similarly for inr. The other rules in Figure 5.1 are

self-explanatory.

In addition to the INTRO rules given in Figure 5.1, each data type has an associated set of INTRO rules which describe the semantics of the type. For example, two INTRO rules for the list type are as follows:

$$\frac{}{\ldots \vdash nil : A\ list}$$

$$\frac{\ldots \vdash h : A \qquad \ldots \vdash t : A\ list}{\ldots \vdash h::t : A\ list}$$

The first INTRO rule says that *nil* is a witness for the polymorphic list type. The second rule says that $h :: t$ is a witness for the polymorphic list type if t also is and if, in addition, h is a witness for the type which the list is parameterized over. From the programming perspective, these rules give a way of constructing any list.

ELIM rules

These rules manipulate the hypothesis list, e.g. to invoke an induction on some variable(s) hence providing new hypotheses. I give the ELIM rule for logical conjunction:

$$\frac{\ldots\ ,\ X_a : A\ ,\ X_b : B\ \vdash\ c : C}{\ldots\ ,\ P :\ A \wedge B\ \vdash\ spread(P, \lambda X_a : A.\lambda X_b : B.c) : C}\ \text{E-}\wedge$$

This rule says that if we have a program to compute C that uses programs that compute A and B, then we can construct a program $spread(P, \lambda X_a : A.\lambda X_b : B.c)$ to compute C that uses any program P that computes A and B. *spread* is a function that takes a pair and a lambda expression. On execution, *spread* returns the lambda expression with the arguments substituted by the components of the pair —
i.e. $spread(< X, Y >, \lambda X_a.\lambda X_b.c(X_a, X_b)) = c(X, Y)$.

Look at the rule E-$\vee$ in Figure 5.2. *decide* is a function which has three arguments. If the first argument is of the form $inl(\ldots)$, it returns its second argument. Otherwise, it returns its third argument. The rule says that if we can prove C using a proof of A and we can prove C using a proof of B, then we can prove C using a proof of $A \vee B$. This new proof will depend on what the proof of $A \vee B$ looks like.

Again, there are ELIM rules for each datatype. An ELIM rule for lists is as given below:

$$\frac{\ldots \vdash a_b : A(nil) \qquad \ldots H : B\ ,\ T : B\ list\ ,\ X_a : A(T) \ldots \vdash a_s : A(H :: T)}{\ldots L : B\ list \ldots \vdash list\_ind(L, a_b, \lambda H.\lambda T.\lambda X_a.a_s) : A(L)}$$

The rule says that to prove property A for a list L it is enough to show that A holds when L is an empty list and when L is non-empty. In the non-empty case, the proof of the property for $H :: T$ assumes the proof of the property for T. The proof (or program) for arbitrary L is constructed via the function *list_ind* which is defined as follows:

$$list\_ind(nil, a_b, a_s) = a_b \tag{5.1}$$

$$list\_ind(H :: T, a_b, a_s) = a_s(H, T, list\_ind(T, a_b, a_s)) \tag{5.2}$$

Note that this rule is precisely the rule of primitive induction on polymorphic lists with X_a labelling the induction hypothesis.

The Duality of Induction and Recursion

This last example highlights the relationship between induction and recursion. Induction is the proof equivalent of recursion. Properties of recursive programs must be proved using mathematical induction. Some non-primitive recursions may be constructed via multiple applications of primitive inductions. For example, consider the `addlist` function:

```
fun addlist (nil,y) = y
|   addlist (x::xs,nil) = x::xs
|   addlist (x::xs,y::ys) = x+y :: addlist (xs,ys);
```

This function is defined via a recursion over two arguments, so the corresponding induction must also be over two arguments. This induction can be realised in two ways. The first is by two applications of the inference rule for primitive list induction above — one for the first and one for the second parameter. The second possibility is to encode the two inductions as a single inference rule or *derived* rule:

$$\frac{\begin{array}{c}\ldots \vdash a_{b_1} : A(nil, Y) \\ \ldots \vdash a_{b_2} : A(X, nil) \\ \ldots H_1{:}B, H_2{:}C, T_1{:}B\ list, T_2{:}C\ list, X_a{:}A(T_1,T_2) \vdash a_s{:}A(H_1{::}T_1, H_2{::}T_2)\end{array}}{\ldots L_1{:}B\ list, L_2{:}C\ list \ldots \vdash mlist\_ind(L_1, L_2, a_{b_1}, a_{b_2}, \lambda H_1.\lambda H_2.\lambda T_1.\lambda T_2.\lambda X_a.a_s){:}A(L_1, L_2)}$$

where *mlist_ind* is defined by:

$$mlist\_ind(nil, Y, a_{b_1}, a_{b_2}, a_s) = a_{b_1} \tag{5.3}$$

$$mlist\_ind(X, nil, a_{b_1}, a_{b_2}, a_s) = a_{b_2} \tag{5.4}$$

$$mlist\_ind(H_1 :: T_1, H_2 :: T_2, a_{b_1}, a_{b_2}, a_s) = \tag{5.5}$$

$$a_s(H_1, H_2, T_1, T_2, mlist\_ind\ (T_1, T_2, a_{b_1}, a_{b_2}, a_s)) \tag{5.6}$$

The use of ELIM rules to implement recursion is central to the ideas embodied in *CYNTHIA*. Note that the use of the ELIM rule for lists ensures that the corresponding recursion is well-founded. *CYNTHIA* is able to guarantee termination of ML programs by providing corresponding well-founded induction rules. To allow for a greater degree of flexibility than simply combining different ELIM rules, *CYNTHIA* uses induction rules that are checked for termination by a technique known as Walther Recursion — see §6.1.1, p. 114.

DECIDE rules

Although, in general, in a constructive logic, the statement $x = y \vee \neg x = y$ is not true, it can be assumed to be true if a decision procedure exists for deciding whether or not $x = y$. In such a case, a DECIDE rule performs a case-analysis. As an example, since a decision procedure exists if x and y are integers we can include the rule:

$$\frac{\begin{array}{rcl} \ldots X : int, Y : int, V : (X = Y : int) \ldots & \vdash & a : C \\ \ldots X : int, Y : int, V : \neg(X = Y : int) \ldots & \vdash & b : C \end{array}}{\ldots X : int, Y : int \ldots \vdash int\_eq(X, Y, a, b) : C}\ \text{DECIDE}$$

where *int_eq* is a function that returns its third argument if X and Y are equal, according to the decision procedure, and its fourth argument otherwise. $V : (X = Y : int)$ is a statement asserting the fact that X is equal to Y. In program terms, the rule represents a conditional statement. In proof terms, it represents a case-split in the proof. Note that this rule is a specialisation of E-$\vee$ in Figure 5.2.

Cut rule

The Cut rule enables a lemma to be speculated, proved and used within a proof. The Cut rule provides some element of bottom up programming to the logic which allows

a better global structuring of the proof tree. The rule is as follows:

$$\frac{H \vdash a : A \qquad X_a : A, H \vdash c : C}{H \vdash (\lambda X_a.c)(a) : C} \text{ CUT}$$

The Cut rule allows the introduction of an intermediate proof step. It says that to prove C, it is enough to prove some statement A and prove C assuming A. $(\lambda X_a.c)(a)$ will then be a proof of C using the proof of A. A corresponds to a lemma in a proof or an auxiliary procedure in a program.

Program Synthesis

The formalism presented so far is one way to realise the notion of *program synthesis*. To synthesise a function with inputs I and outputs O, set up the following specification:

$$\forall I.\exists O.spec(I, O)$$

where *spec* specifies the function. The sequent representation of this is:

$$\ldots \vdash P : (\forall I : i.\exists O : o.spec(I, O))$$

The idea is that this statement is proved, in a backwards fashion, using the inference rules. This process instantiates P and the final instantiation is the program satisfying the specification. P will be a λ-calculus term. Since the λ-calculus is in effect a functional programming language, the resulting term is a functional program satisfying the specification of the proof. This term is sometimes called an *extract term*. Hence, if the user provides a specification (which may be given to varying levels of detail) and then furnishes a proof of that specification, the extract term of the proof will be a program that is correct with respect to the given specification. $CYNTHIA$ utilises this concept of program synthesis. Users, without knowing it, are in fact editing a proof of a specification (albeit a very weak one). After each edit, a program is extracted which is the program that the user has defined by his edit. See Chapter 6 for more details.

5.2.2 Specifications

In general, specifications may be given at any level of detail. The more detailed the specification, the greater are the guarantees of correctness. I will now give two examples

to illustrate the ideas above and also the different levels of specification. Note that all rules in this section are applied in a backwards fashion.

Weak Specifications

The weakest specifications merely state the existence of an object of a particular type. As an example, consider the following specification:

$$\ldots \vdash P : (\forall X : int. \forall Y : int\ list.\ int\ list) \tag{5.7}$$

The specification (5.7) states the existence of an object which has type $int \rightarrow int\ list \rightarrow int\ list$[1][2].

The specification (5.7) can be used to synthesise any function which has type $int \rightarrow int\ list \rightarrow int\ list$. The particular function synthesised depends on which rules are applied and in which order. Suppose we wish to synthesise the function *delete* that removes a given element, X, from a list of integers, Y. We will apply rules in an order that will synthesise *delete*. The first thing to do in the proof is to apply I-$\forall$ from Figure 5.1 twice. $exp \bullet \{Y/X\}$ means substitute Y for X in exp, so Y is just an instantiation of the universal quantifier. Applying this rule twice, in a backwards fashion, gives a new sub-goal:

$$\ldots X : int, Y : int\ list \ldots \vdash P_2 : int\ list \tag{5.8}$$

where P has been instantiated to $\lambda X : int. \lambda Y : int\ list. P_2$. We now carry out an induction on Y by applying the ELIM rule for lists. This splits the proof tree into two branches, the base and step cases respectively.

$$X : int, Y : int\ list \vdash P_b : int\ list \tag{5.9}$$

$$X : int, Y : int\ list, H : int, T : int\ list, R : int\ list \vdash P_s : int\ list \tag{5.10}$$

After this rule application, P_2 has been instantiated to $list\_ind(Y, P_b, \lambda H. \lambda T. \lambda R. P_s)$. The base case (5.9) is finished by introducing the witness *nil*. This is done by applying

[1] $\rightarrow$ associates to the right.

[2] This specification would be easier to understand if written as $\ldots \vdash P : (\forall X : int. \forall Y : int\ list. \exists Z : int\ list)$ However, the syntax of the logic does not allow us to write this. Instead, we omit Z and interpret (5.7) as meaning the same thing.

the INTRO rule for *nil* which instantiates P_b to *nil*. The value of the witness in the step case depends on the value of H. Hence, we need to carry out a case-split on whether X is equal to H or not. This is done by applying the rule DECIDE($X = H : int$). The proof tree again splits into two branches — one for the case $X = H$ and one for $\neg X = H$ respectively:

$$X : int, Y : int\ list, H : int, T : int\ list, R : int\ list, \ldots$$
$$\ldots v : X = H : int \vdash P_{s_1} : int\ list \tag{5.11}$$

$$X : int, Y : int\ list, H : int, T : int\ list, R : int\ list, \ldots$$
$$\ldots v : \neg X = H : int \vdash P_{s_2} : int\ list \tag{5.12}$$

P_s is now instantiated to $int\_eq(X, H, P_{s_1}, P_{s_2})$. The proof is completed by introducing witnesses — INTRO(R) in (5.11) and INTRO($H :: R$) in (5.12). These INTRO rules instantiate P_{s_1} to R and P_{s_2} to $H :: R$. The proof tree is given in Figure 5.3.

The rules so far represent the program that we wish to extract. To ensure type correctness, the witnesses must be checked to be well-formed (i.e. semantically valid). The facts that *nil*, R and $H :: R$ are well-formed and of type *int list* follow directly from the hypotheses.

The extract term is obtained by collecting together all the instantiations:

$$\lambda X : int.\lambda Y : int\ list.\ list\_ind(Y, nil, \lambda H.\lambda T.\lambda R.int\_eq(X, H, R, H :: R))$$

The extract term could be translated into the ML program:

```
fun delete (x:int) nil = nil
|   delete x (h::t) = if x=h then delete x t
                      else h :: (delete x t);
```

This translation would work as follows. λX and λY tell us that the function has two arguments. The first argument of *list_ind* tells us that pattern matching is required over Y. Hence, two cases are provided in the program — one for when Y is empty and one non-empty. The second argument of *list_ind* gives the result in the base case. The presence of *int_eq* tells us that in the step case we need a conditional statement on whether $X = H$. Since R, by the definition of *list_ind*, represents the recursive call

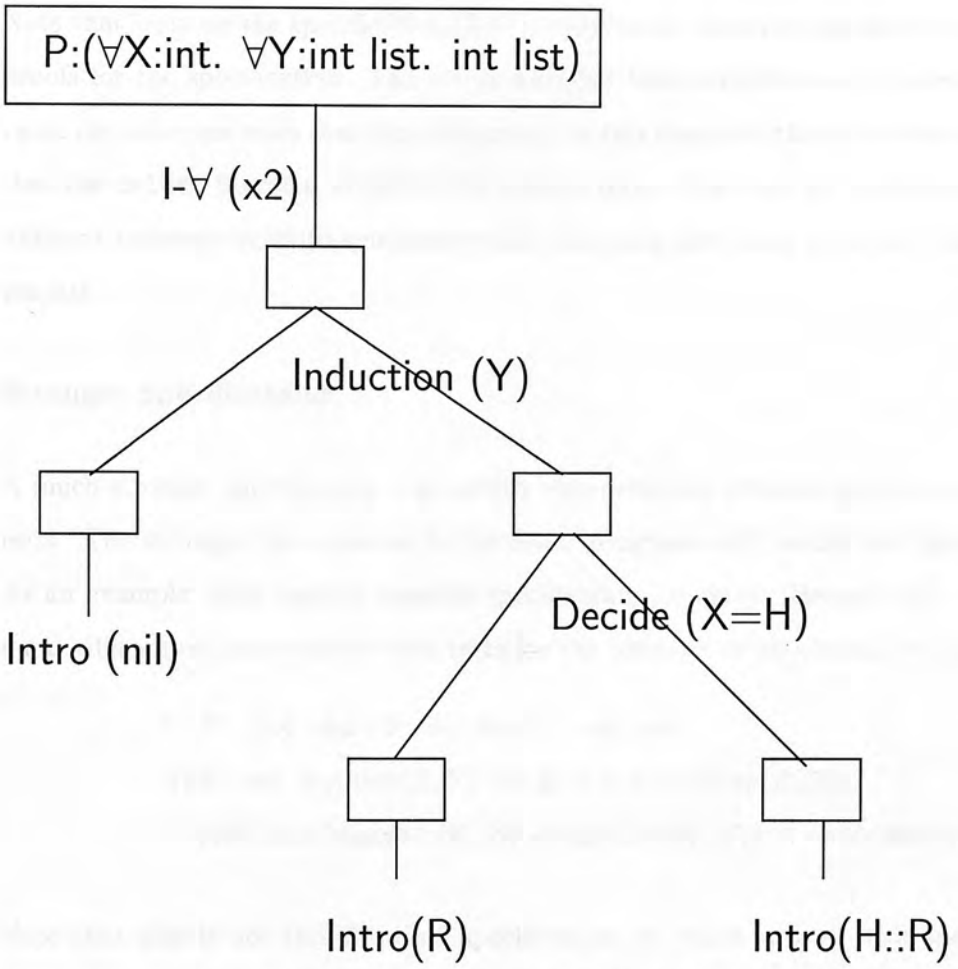


Figure 5.3: Weak Specification Proof Tree.

(i.e. `delete x t`), the two results R and $H :: R$ translate into `delete x t` and `h :: (delete x t)`.

Specifications such as (5.7) provide only limited guarantees of correctness. First, the extract term is guaranteed to satisfy the type given in the specification. In addition, however, the function will be terminating and well-defined. In ML, this translates to mean that if a function is defined by pattern matching, the patterns do not overlap and exhaustively cover the domain of the function. These latter guarantees come from the ELIM rule. As noted before, ELIM rules are well-founded and there will always be precisely one branch of the proof tree for each constructor of the datatype. The final guarantee is syntactic correctness since all terms in the proof must be syntactically valid.

Note that because the specification (5.7) is very weak, there are an infinite number of proofs for the specification. The actual function that is synthesised depends entirely upon the inference rules that form the proof. In this example, the rules were applied so that the `delete` function would be the extract term. However, we could have applied different inference rules to synthesise other functions that have type $int \rightarrow int\ list \rightarrow int\ list$.

Stronger Specifications

A much stronger specification is to specify some relation between the inputs and outputs. The stronger this relation is, the fewer programs will satisfy the specification. As an example, take another possible specification for *delete* [Hesketh 95]. This uses an auxiliary function *member* that tests for the presence of an element in a list.

$$\begin{aligned}\ldots \vdash\ & P : (\forall X : int\ \forall Y : int\ list\ \exists Z : int\ list. \\ & (\forall E : int.\ member(E, Y) \rightarrow (E = X \vee member(E, Z))) \\ & \wedge (\forall W : int.\ member(W, Z) \rightarrow member(W, Y)) \wedge \neg member(X, Z))\end{aligned}$$

Note that this is not the strongest specification we could have — this specification allows the result, Z, to be a permutation of the elements of Y with X removed. The first steps of the proof follow those in the proof of the weak specification. I-$\forall$ is again applied twice, and then induction on Y takes place to give two subgoals:

$$\begin{aligned}& X : int \\ & Y : int\ list \\ & \vdash P_b : (\exists Z : int\ list.\ (\forall E : int.\ member(E, nil) \rightarrow (E = X \vee member(E, Z))) \\ & \quad \wedge (\forall W : int.\ member(W, Z) \rightarrow member(W, nil)) \wedge \neg member(X, Z))\end{aligned}$$

$$\begin{aligned}& X : int \\ & Y : int\ list, H : int, T : int\ list \\ & R : \exists Z : int\ list.(\forall E : int.\ member(E, T) \rightarrow (E = X \vee member(E, Z))) \\ & \quad \wedge (\forall W : int.\ member(W, Z) \rightarrow member(W, T)) \wedge \neg member(X, Z) \\ & \vdash P_s : (\exists Z' : int\ list.(\forall E : int.\ member(E, H :: T) \rightarrow E = X \vee member(E, Z')) \\ & \quad \wedge (\forall W : int.\ member(W, Z') \rightarrow member(W, H :: T)) \wedge \neg member(X, Z')\end{aligned}$$

Compare the hypothesis R to that in (5.10). The base case is continued in essentially the same way as with the weak specification — I-$\exists$ is applied with t instantiated to *nil* to remove the existential quantifier. This gives the subgoals: $\ldots \vdash nil : A$ and $\ldots \vdash b : (\ldots)$. The step case, however, is different. We have to be able to strip off the existential quantifier in R. To do this, we apply E-$\exists$ (Figure 5.2) which gives rise to another hypothesis, R_1, identical to R except that it has no $\exists$ (the rule introduces a new variable Y but in this example we can, wlog., just rename Y to X). The rest of the step case mirrors the weak specification proof — there is a case-split followed by applications of INTRO.

Unlike in the case of weak specifications, however, this is not the end of the story. The introduction of the existential witnesses gives rise to further subgoals which themselves have to be proved (and this can be non-trivial). For instance, in the base case, we have to prove:

$$\begin{aligned} \ldots \vdash\ & (\forall E : int.\ member(E, nil) \rightarrow E = X \vee member(E, nil)) \\ & \wedge (\forall W : int.\ member(W, nil) \rightarrow member(W, nil)) \ \wedge \neg member(X, nil) \end{aligned}$$

where *nil* has been substituted for Z.

This example illustrates that there are two distinct parts to a synthesis proof — the synthesis part and the verification part. The synthesis part is the part which encodes the `delete` function in the proof by introducing existential witnesses. The verification part is the non-computational part which proves that the function satisfies the behaviour stated in the specification. Note that synthesis proofs with the weakest form of specification have a verification part consisting of merely well-formedness checking.

The type of proofs that *CYNTHIA* uses have a weak form of specification — the specification states merely the existence of an object of a certain type plus bounding lemmas about the definition needed in the termination analysis. There are a number of reasons for this. First, the aim of *CYNTHIA* is to be a programming environment, not a proof environment. The aim of using type theory is as a framework for proving relatively simple properties about users' programs in order that their speed of programming is increased because they are restricted from making certain kinds of errors. Users of *CYNTHIA* are not concerned with proving properties of programs *themselves*.

Other systems exist for this purpose — for example, Extended ML [Sannella 90] — but the burden of proof rests in the main part with the user. *CYNTHIA* is not intended to be a system where the user is expected to know about type theory and to be able to carry out proofs. Hence, the program analyses that *CYNTHIA* undertakes are simple enough to be performed automatically. By restricting the synthesis proofs to weak specifications, the proofs become simple enough to be proven automatically in a reasonable amount of time. Of course, the user must still provide the witnesses of the proof but the guarantees of correctness that *CYNTHIA* provides can be analysed without further user interaction. An interesting project would be to consider how to extend *CYNTHIA* to deal with more detailed specifications. This is beyond the scope of this thesis but some preliminary remarks are presented in Chapter 10.

5.3 The *Oyster* Proof Checker

Oyster is an interactive proof checker, developed at Edinburgh, for proofs in a variant of Martin-Löf's Type Theory. It is a re-implementation of the Nuprl [Constable *et al.* 86] system built at Cornell University. *Oyster* can be used to synthesise programs in the manner described earlier in this chapter. *Oyster* works in a goal-directed fashion. A goal is gradually refined by the application of inference rules to $\ldots \vdash true$. Each type in *Oyster* has a number of built-in constructors for the creation of terms of that type (in the form of INTRO rules) and selectors for constructing terms of other types out of the terms of the type under consideration (in the form of ELIM rules). The basic types available in *Oyster* are *atom*, *nat* (providing the natural numbers with Peano arithmetic), *void* (the empty type), and *int*. Derived types include finite lists, function types, product types, quotient types, set types and recursive types. In addition, three relations can be specified — namely, membership, equality and orderings on integers and natural numbers. User-defined types can be given in terms of the basic and derived types. The type structure of *Oyster* is much richer than that of any programming language and corresponds to a more mathematical way of building data structures.

5.3.1 Tactics in *Oyster*

Oyster implements the inference rules given in §5.2, although the notation used is slightly different. The specification

$$\ldots \vdash P : (\forall X : int. \forall Y : int\ list.\ int\ list)$$

would be represented in *Oyster* as:

$$\ldots \vdash \texttt{int} \rightarrow \texttt{int list} \rightarrow \texttt{int list}$$

This states the existence of a function with type $int \rightarrow int\ list \rightarrow int\ list$[3] The variable names X and Y can be introduced as parameters to INTRO rules.

In *Oyster*, the user is provided with a way of combining inference rules into *tactics*. Tactics allow the user to compose common sequences of inference rules (or other tactics) into a single unit. Tactics are applied in exactly the same way as inference rules. The result of a tactic application is the result of applying the rules that make it up. The way of combining rules or tactics is by using *Oyster*'s predefined tacticals. The only two tacticals I will be concerned with in this thesis are *then* and *repeat*. X *then* Y applies tactic X and then Y to the subgoal produced by the application of X. If X produces more than one subgoal, Y should be a list of length the same as the number of subgoals produced by X. *repeat* X applies X and then iteratively applies X to the new subgoals until X can no longer be applied.

Chapter 6 explains how ML programs can be encoded in *Oyster* as synthesis proofs. I give here, however, some remarks that justify the use of *Oyster* as the underlying proof implementation. It should be noted that there are a number of significant differences between the type system of ML and that of *Oyster*. These are:

- The syntax of types in ML and *Oyster* are different, but not in a significant way. For instance, the product of two integers is represented as `int * int` in ML and `int # int` in *Oyster*. A further complication is that the precedences of type constructors in ML is different from that in *Oyster*. This is not a problem,

[3] Like ML, *Oyster* implements both curried and uncurried definitions.

however, as each type expression is first translated to an abstract syntax tree and from there, may be converted into the required type system.

- *Oyster* does not support some types that are present in ML — most importantly, real numbers, characters and record types. Record types have not been fully implemented in *CYNTHIA* because they are not usually required in a first course in ML.

- *Oyster* has a much more expressive type system. This means that whereas the ML type system is decidable, *Oyster*'s is not. Type-checking in *Oyster* is done by defining tactics to check that an expression inhabits a certain type. The undecidability of type checking in *Oyster* is not a problem in practice because *CYNTHIA* restricts to types that correspond to ML types, anyway. The undecidability is only an issue when dealing with types outside this restriction.

- *Oyster* does not have type inference. This is not a problem because *CYNTHIA* requires a top-level type declaration anyway, as explained in Chapter 4, on the grounds that it makes students more aware of types. Chapter 10 discusses how type inference could be incorporated into *CYNTHIA*.

- Polymorphism is not fully supported in *Oyster*. Polymorphism can easily be encoded in a specification as follows:

  ```
  'a:u(1) -> 'a list -> 'a
  ```

  `'a:u(1)` essentially means "`'a` is some type". Polymorphism is not supported in the type-checking tactics that come with the *Oyster* package. Hence, these tactics were extended to carry out the substitutions necessary to deal with polymorphism.

- Mutually recursive functions cannot be defined in *Oyster*. To include mutual recursion in *Oyster* would require extending the logic. This has not been done and so *CYNTHIA* does not support mutually recursive definitions. *CYNTHIA* users can make mutually recursive definitions if they utilise the following trick. To define mutually recursive functions f and g, we instead define a function fg which computes the pair $(f\ x, g\ x)$. To illustrate briefly, consider this example. One way to define `even` and `odd` functions over the natural numbers is via mutual recursion as follows:

```
fun even zero = true
|   even (s n) = odd n
and
    odd zero = false
|   odd (s n) = even n;
```

Using the "trick", this becomes:

```
fun evenodd zero = (true,false)
| evenodd (s n) = let val r = evenodd n in (snd r,fst r) end;
```

with `even` (and similarly `odd`) now defined by

```
fun even n = fst (evenodd n);
```

- User-defined types are not yet supported in *CYNTHIA*. They are supported in *Oyster*, however, so to include them in *CYNTHIA* would require a translation mechanism between ML `datatype` declarations and *Oyster* type definitions. Note, however, that the current mechanisms for defining types in *Oyster* are awkward and lengthy.

Despite these drawbacks, it was considered the type systems are sufficiently similar to enable a relatively simple translation algorithm between the two so that the user deals only with ML types and the program analysis modules deal only with *Oyster* types. In addition, it is possible to define tactics in *Oyster* which directly mimic ML operations. For instance, a tactic can be defined corresponding to a conditional statement. This means that the translation between proofs and programs is simplified and there is a direct mapping between positions in the program, where the user may apply an edit, and positions in the proof tree. See Chapter 6 for more details.

The *Oyster* system is a mature system that has been used over a number of years. The alternative would have been to build a new proof implementation with a logic closer to that of ML or even to embed the formal semantics of ML [Milner *et al.* 90] into a theorem prover. However, this would have been a major undertaking and although conceptually appealing would have had little effect on the finished appearance and functionality of *CYNTHIA*, although an embedding would probably be a lot slower than the current implementation. Note that a similar approach to mine has been followed by the developers of Extended ML. Proofs about Extended ML programs are carried out in the PVS theorem prover [Owre *et al.* 96]. The programs are represented in PVS by translation rather than an embedding of the semantics of Extended ML within PVS.

5.4 Summary

This chapter has described the technique known as proofs-as-programs which details a correspondence between functional programs and proofs in a constructive logic. *CYNTHIA* uses this concept to represent ML programs and to reason about properties of those programs.

 suppose you've come to collect some knowledges," said Cynthia as she led Mavis away from Walter's cottage.

"Yes," replied Mavis. "I need to take some back to my own island. Knowledges are really sought after over there. And anyone whose got some, earns great respect from the Elders."

Knowledges are small, azure-coloured jewels. Of all the islands in the world, there are very few places where knowledges can still be found. One of these places is MikaLand. Rumour has it that at the centre of the forest lies a great open expanse, with an enormous castle right in the middle of it. Travellers tell of the halls of this castle being paved in knowledges. Many people native to Mavis's village had come to MikaLand in search of these knowledges, but most had returned empty handed. Or worse still, not returned at all.

"Well," said Cynthia. "You're going to need some serious help if you want to find your way to the castle."

"Will you help me?" inquired Mavis hopefully.

"Of course," cried Cynthia. "Why do you think I'm here?"

Chapter 6

ML Programs as Synthesis Proofs

This chapter explains in detail how ML Programs can be encoded as synthesis proofs. Inference rules necessary to describe ML programming constructs and correctness-checking are given along with a discussion of how they were implemented. An analogical mechanism is presented which details how synthesis proofs are modified in reaction to users' edits on ML programs. An important part of the analysis is to show termination of the programs edited in *CYNTHIA*. This chapter explains Walther Recursion, a technique which enables a wide range of functional programs to be shown terminating. Walther Recursion is presented for a small functional language in [McAllester & Arkoudas 96]. I have extended its use to ML programs as accepted by *CYNTHIA*.

The idea behind *CYNTHIA* is to exploit the close correspondence between synthesis proofs and ML programs. This chapter details how ML programs can be encoded as synthesis proofs. These synthesis proofs are implemented in the Edinburgh proof editor *Oyster*. The basic idea is that each programming construct in the ML language corresponds to an inference step in an accompanying logic. In addition, the synthesis proof will contain subproofs dealing with the correctness of the program.

The *Oyster* system [Bundy *et al.* 90] is a reconstruction of Nuprl [Constable *et al.* 86] which accommodates the top-down construction of proofs by refinement. Nuprl supports a variant of Martin-Löf's Type Theory [Martin-Löf 79]. It is implemented in Lisp

/ ML whereas *Oyster* is implemented in Prolog. In *Oyster*, the root of the proof is a goal specification which is gradually refined by the application of inference rules. As explained in Chapter 5, all rules are applied in a backwards-fashion. Given the goal to be proved, an inference rule is applied to refine the goal[1]. This process may produce further subgoals. In this chapter, inference rules will be presented as follows:

$$\frac{\begin{array}{c} H_1 \vdash T_1 \\ \vdots \\ H_n \vdash T_n \end{array}}{H \vdash T} \text{ RULENAME}$$

The goal is given below the horizontal line and the subgoals above. The rule should be read as — T can be shown under assumptions H if each of T_i can be shown under H_i.

Proof editors such as *Oyster* have a collection of low-level inference rules that each contribute a very small part to the proof. This is so that the editor is as general as possible. To enable practical theorem proving, however, *tactics* can be defined which automate a sequence of inference rule applications. By a process known as grafting [Richards 93], tactics may appear in the proof tree as a single unit so that the constituent inference rules are hidden. In this way, the proof tree appears as a tree of tactic applications and is therefore clearer and easier to understand.

Tactics turn out to be a very convenient way of defining ML programs as synthesis proofs. Tactics can be written to represent each programming construct in ML and to represent correctness-checking routines such as type-checking and termination-checking. Tactics are a natural way to achieve the correct level of abstraction within the synthesis proof. Abstraction is important because:

- Producing the complex editing commands described in Chapter 4 requires an analogical mechanism that will propagate isolated changes throughout a synthesis proof. For example, suppose the user applies the CHANGE TYPE command to change the result type in his program. This corresponds to a change in the specification of the synthesis proof which must be modified to reflect the new

[1] Throughout the rest of this thesis, refinement refers to backwards proof and all inference rules are applied in a backwards-direction.

type. However, this process yields an incorrect proof because the proof is based on the previous specification. Analogy can be used to propagate the effect of CHANGE TYPE throughout the proof. The mechanism attempts to resolve any inconsistencies in the proof. Analogy becomes easier and more successful when the level of abstraction is raised [Melis & Whittle 98]. At too low levels, immense effort is required to produce a proof by analogy. Abstraction means that minor details can be ignored. It is these minor details that are most likely to be responsible for the breakdown of the analogy.

- Abstraction simplifies the process of extracting programs from proofs. Each node in the proof tree has an associated extract term. Obtaining the extract term for a completed proof involves combining the extract terms at each node. During this process, tactics are unpacked into the sequence of rule applications which make them up. Hence, the extract term will not normally retain the same level of abstraction but will consist of a chain of primitive expressions which would then have to be reconstituted into an ML program. This process can be simplified by avoiding the unpacking of the tactics and associating each tactic with an extract term more directly. *CYNTHIA*'s tactics are designed in such a way as to make the extract terms resemble as closely as possible the intended ML program fragment. Hence, the tactic for introducing a casesplit in the proof has an extract term `if then else` which corresponds exactly to an ML construct. In this way, we retain the same level of abstraction which greatly simplifies the translation from proof to program.

- A good deal of ambiguity arises when translating between synthesis proofs and ML. On the one hand, different ML constructs may correspond to the same proof step. For example, both constructs `let fun` and `let val`[2] are represented in proofs by introducing a lemma that asserts the existence of an object of a certain type. If both constructs are represented by the same inference step in the proof, it is impossible to know which construct to use when the proof is translated back into a program. On the other hand, different synthesis proofs may correspond to the same code. For example, to solve a subgoal $\vdash$ `int * int`,

[2] These are compound constructs in ML of the form '`let <decl> in <expr> end`' where '`decl`' could be '`fun ... = ...`' or '`val ... = ...`'.

the object `(0,0)` could be introduced. Alternatively, the INTRO rule for `*` could be applied yielding two identical subgoals: $\vdash$ `int`. These could be resolved by introducing `0` in both cases. These ambiguities confuse the translation between program and proof. To overcome this, tactics are used to restrict the proofs that can be created. The user may only construct proofs using the tactics provided and these tactics have been designed to avoid any of the ambiguities noted above.

To summarise, tactics have been designed to represent ML programs as proofs. There are three types of tactics:

1. Tactics corresponding to ML constructs — one for each of `if then else`, `case`, `let fun`, `let val` and `fn`.

2. Tactics corresponding to programming concepts in ML — e.g. a tactic which allows pattern matching and recursion, and a tactic which is used to specify a definition's output (or witness).

3. Tactics for correctness-checking — tactics for termination checking, WFFTACS[3] for type-checking and a tactic for checking static semantics, SYNTACS.

The use of the above tactics restricts the proof in such a way as to eliminate the ambiguities mentioned previously. The proof trees will have a form something like that given in Figure 6.1. The specification is represented by an elliptical node. Square nodes correspond to tactics of type (1) and (2) above, and triangular nodes correspond to type (3) tactics. The diagram is meant to represent the form of a typical proof tree but there are exceptions. In particular, a function need not be defined by recursion — in this case, there will be no recursion tactic.

When tactics are applied in a proof, the tactic is expanded so that each inference rule that makes it up is applied. Unfortunately, this process can be time-consuming because each tactic may contain a large number of inference rules. Efficiency is an important issue surrounding *CYNTHIA* because it is meant to be a usable system. It turns out that tactics of type (1) and (2) can be simplified considerably by factoring out a lot of the simple inference rules. Typically, an *Oyster* tactic will contain general-purpose

[3] WFFTACS was already available as part of *Oyster* but has been modified in *CYNTHIA*.

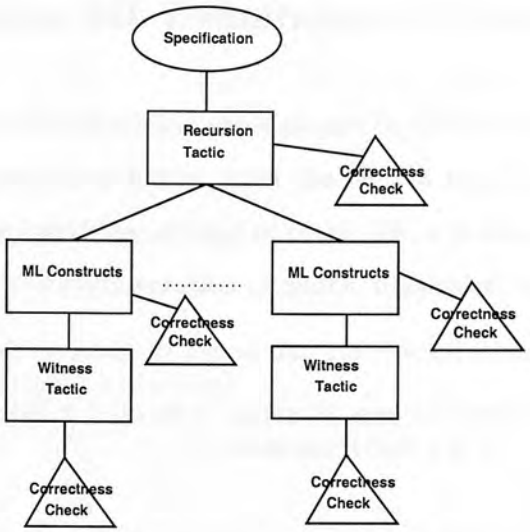


Figure 6.1: Typical Form of Proof Trees in *CYNTHIA*

normalisation routines, such as a routine for decomposing a pair. The normalisation is done by applying a sequence of rules. The tactics in *CYNTHIA* do not have such normalisation processes. Instead, the main part of the tactic is actually implemented as an inference rule. Normalisation is achieved by a piece of Prolog code attached to this rule. This piece of code carries out the same algorithm as the original inference rules. However, this latter approach is more efficient. Applying an inference rule involves routines for inserting, deleting or replacing parts of the proof tree, along with book-keeping tasks that go with it. By factoring out the normalisations, fewer inference rules are being applied and hence less work is carried out on tree maintenance. The down side is that by replacing rules by Prolog code there is a risk of loss of correctness. However, this is a small risk as the Prolog code is well understood and corresponds directly to inference rules.

To summarise, the rules presented in this chapter extend *Oyster*. They could have been (and indeed originally were) written as tactics but were written as rules for reasons of efficiency. Throughout the rest of this thesis, I shall use the terms tactic and rule interchangeably.

6.1 Representing ML Definitions as Proofs

This section presents the underlying proof engine in *CYNTHIA*. Note that all the theorem proving is completely hidden from the user so that the user of *CYNTHIA* requires no specialised knowledge of logic or proof. We will use an ongoing example to illustrate the ideas — the representation of `qsort`, illustrated in Fig. 6.2.[4]

```
(int * int -> bool) -> int -> int list -> int list
fun partition f k nil = nil
|  partition f k (h::t) = if f(h,k) then h::partition f k t
                          else partition f k t;


int list -> int list
fun qsort nil = nil
|  qsort (h::t) = (qsort (partition (op <) h t)) @ [h]
                        @ (qsort (partition (op >=) h t));
```

Figure 6.2: A Version of Quicksort.

6.1.1 Walther Recursion

§5.2 described the ideas behind the concept known as proofs-as-programs. In synthesis proofs, termination of recursive programs is guaranteed by the application of a well-founded induction scheme. Unfortunately, this means that the user is limited to a pre-defined library of induction schemes or must prove the well-foundedness of any new scheme that is introduced. This section describes the technique Walther Recursion [McAllester & Arkoudas 96] which checks well-foundedness for a subset of the set of terminating programs. §6.1.2 and §6.1.3 detail how Walther Recursion has been incorporated into the proofs-as-programs framework.

Termination is a central problem in automated reasoning. Many logics, such as that of Boyer and Moore [Boyer & Moore 79], assume that functions are total. Even in systems that deal with partial functions such as IMPS [Farmer *et al.* 93], termination is still important. For example, proving a lemma such as $\forall x\ f(x) > x$ requires a termination proof for f. Termination has been studied extensively.

[4] `::` is the ML cons operator for lists. `@` is append.

The approach taken by Boyer-Moore is to base termination proofs on user-defined orderings. Although this is a general approach, it requires the user to specify an ordering and also to show that the ordering is well-founded. In a tool such as *CYNTHIA*, the user is expected to have no knowledge of how to go about such proofs. Hence, this approach is insufficient. An alternative would be to restrict the user to primitive recursive definitions. This is the approach taken by other editors — for example, the recursion editor [Bundy *et al.* 91] restricts to primitive recursion with a few, minor extensions. In general, however, the user will want to write many definitions which are not primitive recursive. Hence, this alternative is insufficient.

Walther [Walther 88b] presents a calculus for ensuring measure decreases over recursive calls under a fixed size measure. Walther's calculus is unsuitable for *CYNTHIA*, however, as it is undecidable. McAllester and Arkoudas [McAllester & Arkoudas 96] describes a decidable restriction of Walther's calculus suitable for termination analysis in *CYNTHIA*. The restriction is such that termination analysis may be carried out automatically in a reasonable amount of time. Hence, the user need not provide orderings or proofs of well foundedness. Of course, there are terminating functions that cannot be defined in McAllester's restriction, but the range of functions that can be defined is significantly wider than the set of functions that satisfy the criterion for primitive recursion.

I present McAllester's technique here, known as Walther Recursion. McAllester's analysis is actually for a small, hypothetical functional language but I have modified his technique to work for ML function definitions. It is this modification that I present here. I assume that the analysis is applied to a complete set of definitions. §6.2 considers how this has been modified to fit the framework of *CYNTHIA* whereby analysis is done incrementally for possibly incomplete programs.

Before I describe the technique, however, I want to give an idea of the sort of functions that are Walther Recursive. It is difficult to provide a precise characterisation, but the following kinds of functions are typical:

- Primitive recursive functions over an inductively-defined datatype. For example:

```
fun count (leaf n) = 1
|   count (node(n,x,y)) = 1 + count x + count y;
```

These include functions that recurse with accumulators:

```
fun rev nil y = y
|   rev (h::t) y = rev t (h::y);
```

as well as higher-order functions that satisfy the fixed size ordering, such as `partition` in Figure 6.2.

- Multiple recursive functions — i.e. where there is more than one recursive argument. The function `addlist` given in §5.2 is an example of a function with multiple recursion:

```
fun addlist (nil,y) = y
| addlist (x::xs,nil) = x::xs
| addlist (x::xs,y::ys) = x + y :: addlist (xs,ys);
```

- Nested recursive functions — where recursive calls appear inside other recursive calls.

```
datatype lambda = Var of string | Apply of lambda * lambda
                                | Lambda of string * lambda;

val freshvar = "z";

fun rename x y (Var w) = if x=(Var w) then y else x
|   rename x y (Apply(l,r)) = Apply(rename x y l, rename x y r)
|   rename x y (Lambda(w,l)) = Lambda (freshvar,
                  rename x y (rename (Var w) (Var freshvar) l));
```

- Functions defined by non-primitive patterns:

```
fun even zero = true
|   even (s zero) = false
|   even (s (s n)) = even n;
```

- Functions that reference previously defined functions in a recursive call, such as `qsort`, in Figure 6.2.

- Combinations of the above.

The Measure used in Walther Recursion

There are two parts to Walther Recursion — reducer / conserver (RC) analysis and measure argument (MA) analysis. Every time a new definition is made by the user,

reducer / conserver lemmas are calculated for the definition. These place a bound on the definition based on a fixed size ordering. To guarantee termination, it is necessary to consider each recursive call of a definition and show that the recursive arguments decrease with respect to this ordering. Since recursive arguments may in general involve references to other functions, a measure decrease is guaranteed by utilising previously derived RC lemmas. The fixed size measure, w, used throughout the analysis is defined by the rules given in Figure 6.6, p. 128. Intuitively, this measure could be defined by:

$$w(c(u_1, \ldots, u_n)) = 1 + \sum_{i \in R_c} w(u_i)$$

where c is a data constructor and R_c is the set of recursive arguments of c. If $c(u_1, \ldots, u_n)$ has type T then the recursive arguments of c are the i such that u_i also has type T. For example, given the constructor term `h::t`, then `t` has the same type as `h::t` and is hence a recursive argument. The distinction between reducer and conserver lemmas is given as follows. First, define the semantics of the inequality operator.

Definition 2 *Define $u \leq_w t$ if the following conditions hold:*

- *If u is well typed then t is well typed.*
- *If u is well typed then the top level constructor of u is either a base constructor or the same as the top level constructor of t.*
- *If u is well typed then the measure of u is no larger than the measure of t.*

Define strict inequality in a similar way.

Definition 3 *A constructor is a step constructor if at least one of its arguments is recursive, and is a base constructor otherwise.*

Definition 4 *A function f is a reducer on its ith argument if*

$$f\ x_1 \ldots x_n <_w x_i \tag{6.1}$$

and a conserver on its ith argument if

$$f\ x_1 \ldots x_n \leq_w x_i \tag{6.2}$$

To simplify the analysis, $<_w$ can be eliminated by rewriting (1) as:

$$f\ x_1 \ldots c_j(\ldots, r_{j,k}, \ldots) \ldots x_n \leq_w r_{j,k} \tag{6.3}$$

where c_j is a constructor and $r_{j,k}$ is a recursive argument of c_j. This means that only one form of inequality is ever present.

Throughout the rest of this chapter, the function f will be given in curried fashion. *CYNTHIA* accepts either curried or uncurried definitions.

Reducer / Conserver Analysis

This will be illustrated by example. Consider the following possible definition of a `replace` function:

```
fun replace(x,y,nil) = nil
 |  replace(x,y,h::t) = if h=x then y :: t
                                 else h :: replace(x,y,t);
```

We can derive a conserver lemma for `replace`, namely: `replace(x,y,z)` $\leq_w$ `z`. It is enough to show that the lemma holds for each possible output of the function. Using our intuitive notion of w, `nil` $\leq_w$ `nil`. And, using an induction argument, we can show the result for `h::t`. If `replace(x,y,t)` $\leq_w$ `t`, then w(`h::replace(x,y,t)`) = 1+ w(`replace(x,y,t)`) $\leq$ 1+w(`t`). Since, w(`h::t`) = 1+w(`t`), then `h` :: `replace(x,y,t)` $\leq_w$ `h` :: `t` and so the lemma holds for the case ¬`h` = `x`. For the case `h=x`, `y` :: `t` $\leq_w$ `h` :: `t` holds by our intuition about w. Hence, the conserver lemma is proved.

Measure Argument Analysis

The idea behind measure argument analysis is to show that the measure decreases over each recursive call. It is not sufficient merely to consider each recursive argument in isolation, however. Consider the following function definition:

```
fun loop nil y = nil
|   loop x nil = x
|   loop (x::xs) (y::ys) = if x=y then loop xs (x::y::ys)
                                  else loop (y::x::xs) ys;
```

This function does not terminate for all arguments — consider `loop [1,2] [1,1]`, for example. However, each recursive call decreases the size of one of its arguments. The notion of measure argument is introduced to overcome this problem.

Definition 5 *Given a function f, defined over arguments $x_1, \ldots, x_n$, the set of measure arguments is the set of i such that for every recursive call $f\ u_1 \ldots\ u_n$ of f, $u_i \leq_w x_i$.*

Measure argument (MA) analysis involves showing that the measure decreases over each recursive call. To check for termination, the procedure in Fig. 6.3 is adopted.

1. Find measure arguments, M, for f by considering each x_i in turn and applying the rules in Fig. 6.6;
2. **if** $M = \{\}$, termination analysis fails.
   else for each recursive call, $f\ u_1 \ldots u_n$, try to find an $m \in M$ such that $u_m <_w x_m$ — i.e. if x_m is a constructor term $c(\ldots, r_j, \ldots)$, we need $u_m \leq_w r_j$ for some j.
   if this can be done for all recursive calls, then f terminates.
   else termination analysis fails

Figure 6.3: Procedure for Checking Termination.

Note that the above definition of `loop` is ruled out because there are no measure arguments. Although each recursive call decreases one argument, the other argument is increased.

Note also that in attempting to derive $u_m <_w x_m$, it may be necessary to use previously derived RC lemmas.

Consider the example `ssort`:

```
fun ssort nil = nil
|   ssort (x::xs) = min(x::xs) :: ssort (replace(min(x::xs),x,xs));
```

`min` returns the minimum of a list of integers. In this example $M = \{1\}$, since `replace(min(x::xs),x,xs)` $\leq_w$ `xs` by our previously derived conserver lemma for `replace`. And `xs` $\leq_w$ `x::xs` yields 1 as a measure argument. Since the inequality is in fact strict, termination is also proved.

It is worth pointing out that for the measure argument analysis to guarantee termination, the function must be defined by a well-defined pattern.

Finally, note that for nested recursions, the RC lemmas required in the measure argument analysis will be precisely those derived for the function definition itself. Consider the nested recursive definition given earlier:

```
fun rename x y (Var w) = if x=(Var w) then y else x
|   rename x y (Apply(l,r)) = Apply(rename x y l, rename x y r)
|   rename x y (Lambda(w,l)) = Lambda (freshvar,
                    rename x y (rename (Var w) (Var freshvar) l));
```

A conserver lemma for `rename` can be derived: `rename x y z` $\leq_w$ `z` (assuming the argument of `Var` is not recursive). Measure argument analysis then needs to use this lemma to show termination.

6.1.2 Specifications

The next two sections describe the representation of ML definitions as proofs. Each ML function is represented by a proof with specification (i.e. top-level goal) that is precisely the type of the function along with RC lemmas required for termination analysis. In general, such specifications may specify arbitrarily complex behaviour about the function. However, *CYNTHIA* specifications are deliberately rather weak so that the theorem proving task can be automated. *CYNTHIA* specifications are defined as follows.

Definition 6 *A CYNTHIA specification of an ML function is of the form:*

$$\begin{aligned} P : (\forall z_1 : T_1 . \ldots \forall z_n : T_n . (f\ z_1 \ldots\ z_n) : T_0\ \wedge \\ (f\ z_1 \ldots z_n) \leq_w z_{i_1} \wedge \ldots \wedge (f\ z_1 \ldots z_n) \leq_w z_{i_r}\ \wedge \\ (f\ z_1 \ldots c_{j_1}(\ldots, r_{j_{1,k}}, \ldots) \ldots z_n) \leq_w r_{j_{1,k}} \wedge \ldots \\ \ldots \wedge (f\ z_1, \ldots, c_{j_s}(\ldots, r_{j_{s,k}}, \ldots) \ldots z_n) \leq_w r_{j_{s,k}}) \end{aligned} \tag{6.4}$$

where:

f represents the name of the function[5]*;*

$T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T_0$ is the type of the function;

P is a variable representing the definition of the ML function. P gets instantiated as

[5] f is given in curried fashion. Again, either curried or uncurried is allowed but I will only give the curried version here.

the inference rules are applied. A complete proof instantiates P to a complete program. See Chapter 5;

$c_{j_1}, \ldots, c_{j_s}$ *are constructors;*

$i_1, \ldots, i_r \in \{1, \ldots, n\}$.

The first part of the specification merely states the existence of a function of type $T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T_0$. Clearly, there are an infinite number of proofs of such a specification. The particular function represented in the proof is given by the user, however, since each editing command application corresponds to the application of a corresponding inference rule. In addition, many possible proofs are outlawed because the proof rules (and corresponding editing commands) have been designed in such a way as to restrict to certain kinds of proofs, namely those that correspond to ML definitions. The second part of the specification states RC lemmas that hold for the function.

In the example, the specification for `partition` is:

$$P : (\forall z_1 : (int * int \rightarrow bool).\ \forall z_2 : int.\ \forall z_3 : int\ list.$$
$$(f\ z_1\ z_2\ z_3) : int\ list\ \wedge\ (f\ z_1\ z_2\ z_3) \leq_w z_3) \qquad (6.5)$$

CYNTHIA specifications are in fact dynamic — in the sense that as edits are applied, the specification may be changed to reflect the modifications.

6.1.3 Inference Rules

Each ML function definition is represented by a proof of the relevant specification. There are three kinds of inference rules used in these proofs.

$$\frac{\begin{array}{rcl} H & \vdash & t : T_0 \\ H & \vdash & t \in \Sigma \\ H & \vdash & A \bullet \{t/(f\ x_1 \ldots x_n)\} \end{array}}{H \vdash t : ((f\ x_1 \ldots x_n) : T_0 \wedge A)} \text{ WITNESS}$$

$$\frac{\begin{array}{rcl} H & \vdash & e_1 : bool \\ H, X : e_1 & \vdash & e_2 : A \\ H, X : \neg e_1 & \vdash & e_3 : A \\ H & \vdash & e_1 \in \Sigma \end{array}}{H \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) : A} \text{ IF}$$

$$\frac{\begin{array}{rcl} H & \vdash & e_1 : T \\ H & \vdash & e_1 \in \Sigma \\ H, v : T, X : (v = e_1) & \vdash & e_2 : A \end{array}}{H \vdash (\texttt{let val}\ (v : T) = e_1\ \texttt{in}\ e_2\ \texttt{end}) : A} \text{ LET VAL}$$

$$\frac{\begin{array}{l} H \vdash e_1 : (\forall v_1 : T_1 \ldots \forall v_n : T_n . (f\ v_1 \ldots\ v_n) : T_0 \wedge (f\ v_1 \ldots v_n) \leq_w v_{i_r} \wedge \\ \qquad \ldots \wedge (f\ v_1 \ldots c_j(\ldots, r_{j_k}, \ldots) \ldots v_n) \leq_w r_{j_k}) \\ H, v_1 : T_1, \ldots, v_n : T_n, f : (T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T_0), (f\ v_1 \ldots v_n) \leq_w v_{i_r}\ , \\ \qquad \ldots, (f\ v_1 \ldots c_j(\ldots, r_{j_k}, \ldots) \ldots v_n) \leq_w r_{j_k} \vdash \\ \qquad\qquad e_2 : A \end{array}}{H \vdash (\texttt{let fun}\ f\ (v_1 : T_1) \ldots (v_n : T_n) = (e_1 : T_0)\ \texttt{in}\ e_2\ \texttt{end}) : A} \text{ LET FUN}$$

$$\frac{\begin{array}{l} H, u_1 : \psi(c_{b_1}, 1) \vdash a_{b_1} : (f(c_{b_1}(u_1)) : T_0 \wedge A(c_{b_1}(u_1))) \\ \vdots \\ H, u_n : \psi(c_{b_n}, 1) \vdash a_{b_n} : (f(c_{b_n}(u_n)) : T_0 \wedge A(c_{b_n}(u_n))) \\ H, v_{11} : \psi(c_{s_1}, 1), v_{12} : \psi(c_{s_1}, 2), \\ f(R_{s_{11}}) : T_0, \ldots, f(R_{s_{1p_1}}) : T_0, \\ X_{11} : A(v_{11}), X_{12} : A(v_{12}) \vdash \\ \qquad a_{s_1} : (f(c_{s_1}(\ v_{11}, v_{12})) : T_0 \wedge A(c_{s_1}(v_{11}, v_{12})) \wedge \\ (R_{s_{11}} \leq_w v_{11} \vee R_{s_{11}} \leq_w v_{12}) \wedge \ldots \wedge (R_{s_{1p_1}} \leq_w v_{11} \vee R_{s_{1p_1}} \leq_w v_{12})) \\ \vdots \\ H, v_{n1} : \psi(c_{s_n}, 1), v_{n2} : \psi(c_{s_n}, 2), \\ f(R_{s_{n1}}) : T_0, \ldots, f(R_{s_{np_n}}) : T_0, \\ X_{11} : A(v_{n1}), X_{12} : A(v_{n2}) \vdash \\ \qquad a_{s_n} : (f(c_{s_n}(\ v_{n1}, v_{n2})) : T_0 \wedge A(c_{s_n}(v_{n1}, v_{n2})) \wedge \\ (R_{s_{n1}} \leq_w v_{n1} \vee R_{s_{n1}} \leq_w v_{n2}) \wedge \ldots \wedge (R_{s_{np_n}} \leq_w v_{n1} \vee R_{s_{np_n}} \leq_w v_{n2})) \end{array}}{\begin{array}{l} L : B \vdash (ind(L, \lambda u_1 . a_{b_1}, \ldots, \lambda u_n . a_{b_n}, \\ \qquad \lambda v_{11} . \lambda v_{12} . \lambda X_{n1} . \lambda X_{n2} .\ a_{s_1}(R_{s_{11}}, \ldots, R_{s_{1p_1}}), \\ \qquad\qquad \vdots \\ \qquad \lambda v_{n1} . \lambda v_{n2} . \lambda X_{n1} . \lambda X_{n2} .\ a_{s_n}(R_{s_{n1}}, \ldots, R_{s_{np_n}}))) : (f(L) : T_0 \wedge A(L)) \end{array}} \text{ IND}$$

$t : T$	t has type T;
$t \in \Sigma$	t is (statically) semantically valid (e.g. no undeclared variables or functions);
$\psi(c, n)$	returns the type of the nth argument of constructor c;
$f(X)$	replace the distinguished argument of f (given by context) by X;
$R_{s_{ij}}$	are the recursive call arguments over which the function is defined;
L	is the induction variable (we restrict to a single induction variable here).

Figure 6.4: Structure Rules for *CYNTHIA* (1).

$$\frac{\begin{array}{rcl} H & \vdash & e : T \\ H & \vdash & e \in \Sigma \\ H, u_1 : \psi(c_1, 1), X_1 : e = c_1(u_1) & \vdash & e_1 : A \\ & \vdots & \\ H, u_n : \psi(c_n, 1), X_n : e = c_n(u_n) & \vdash & e_n : A \end{array}}{\begin{array}{rcl} H & \vdash & (\texttt{case}\ (e : T)\ \texttt{of}\ (c_1(u_1) \Rightarrow e_1 \\ & & \quad \vdots \\ & & \mid c_n(u_n) \Rightarrow e_n)) : A \end{array}}\ \text{CASE}$$

$t : T$	t has type T;
$t \in \Sigma$	t is (statically) semantically valid (e.g. no undeclared variables or functions);
$\psi(c, n)$	returns the type of the nth argument of constructor c.

Figure 6.5: Structure Rules for *CYNTHIA* (2).

Structure Rules

Figs. 6.4 and 6.5 give rules that mirror the structure of the ML definition. Each program construct has a corresponding inference rule. When the user introduces a construct using the editing commands, the appropriate inference rule is applied to the current goal in the proof. As each rule is applied, the variable which represents the program (P) is gradually instantiated.

The WITNESS is similar to the usual INTRO rules. When the rule is applied, t is instantiated to the result of the function for some clause. Hence, t is substituted for $(f\ x_1 \ldots x_n)$ in A. The property A then needs to be proved for this substitution. In addition, t must be shown to be well-typed and (statically) semantically valid.

LET VAL deals with local variable declarations. v and e_1 are instantiated when the rule applies but e_2 remains uninstantiated. The rule introduces equalities into the hypotheses in the third subgoal. These keep a track of variable declarations and are needed for Walther Recursion analysis.

LET FUN introduces a local function into the program. In proof terms, this corresponds to a lemma stating the existence of a function f of type $T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T_0$ satisfying certain RC lemmas. The RC lemmas are chosen in the same way as the RC lemmas in the top-level specification. *CYNTHIA* tries to prove each possible RC lemma but only those that hold remain in the specification. In this way, the specification is dynamic and is updated as edits are applied. The second subgoal assumes the existence of f and the truth of the RC lemmas and tries to instantiate e_2.

IND is a super-rule setting up an induction corresponding to the recursion in the program and also setting up an induction to show the termination of this recursion scheme. $a_{b_1}, \ldots, a_{b_n}$ are base cases. $u_1, \ldots, u_n$ are therefore non-recursive arguments. For the sake of clear presentation, each constructor c_{b_i} is restricted to have only one argument. $a_{s_1}, \ldots, a_{s_n}$ are step cases. Each v_{ij} is a recursive argument. Again, we restrict to just two arguments.

There are two things going on with the IND rule. Firstly, subgoals are set up to carry out measure argument analysis — i.e. check that the recursive calls $R_{s_{ij}}$ are

measure decreasing. This is true as long as each $R_{s_{ij}}$ is measure preserving on a strict subexpression of the pattern over which recursion is defined. Secondly, IND carries out an induction to show that the RC lemmas in the specification hold. The induction scheme is based on the patterns over which the ML function is defined. For each pattern $c_{s_i}(v_{i1}, v_{i2})$, the induction hypotheses state that the property A holds for v_{i1} and v_{i2}.

CASE is self-explanatory. As with LET VAL, equalities are introduced into the hypotheses for Walther Recursion.

Once a proof is completed, the ML program represented by it can be extracted easily. For rules WITNESS, IF, LET VAL and LET FUN, the extract is precisely the instantiation of P. For IND, we need a simple translation from the *ind* function to an ML function definition using patterns.

The second kind of rules are rules for type-checking and checking that a term inhabits Σ. Type-checking subgoals produced by rule applications are proved by the WFFTACS tactic which applies *Oyster*'s standard type-checking (with minor extensions). Similarly, subgoals of the form $t \in \Sigma$ are proved using SYNTACS.

Rules for Walther Recursion

The third kind are rules for Walther Recursion analysis. These are given in Figure 6.6, p. 128. Some of these rules were provided in [McAllester & Arkoudas 96] but are presented here in a form that follows more closely the refinement-style of *Oyster*. The other rules are original.

In [McAllester & Arkoudas 96], Walther Recursion was described for a small functional language with a syntax and semantics different to that of Standard ML. Hence, a number of changes had to be made to account for this.

- **Translation to constructor functions.** McAllester's language is based around the use of destructors for defining functions. For example, the `length` function might be written as:

```
fun length z = if z=nil then 0
```

```
else 1 + length (tl z);
```

Although functions can be defined in this way in ML, it is more natural and common to use pattern matching on expressions defined with constructors. As a result, McAllester's rules need to be recast and an additional rule WCONS3 is added. WCONS3 resolves trivial subgoals such as `t` $\leq_w$ `h::t`. In McAllester's case, WCONS3 is unnecessary as the equivalent subgoal is `(tl z)` $\leq_w$ `z` which is solved by WCONS1 because there would be a lemma stating that `(tl z)` leq_w `z`. See Figure 6.6.

- **Proof Strategy**. When deducing reducer / conserver lemmas, McAllester intends the rules to be applied in a forward direction. In this way, all possible lemmas can be enumerated. The synthesis proofs in *CYNTHIA* are based on a refinement style, so our system sets up subgoals for each possible lemma and then attempts to prove them by applying the rules in a backward fashion.

- **Use of induction**. McAllester's proofs implicitly use induction. In *CYNTHIA*, the connection is made more explicit. All reducer / conserver lemmas are proved by induction.

- **Reducer lemmas**. A function defined by an exhaustive pattern cannot be a reducer. This is because the measure of the base case argument cannot be reduced. For example, if a function is defined by pattern matching on lists, the base case is when the matched parameter is `nil`. The measure of `nil` is minimal and so cannot be reduced. McAllester gets around this by forcing the user to make an additional definition. This new definition is restricted to non-base-cases. For example, the following function is well-defined.

```
fun tl nil = nil
|   tl (h::t) = t;
```

In McAllester's formulation, the user may need to define a restricted definition:

```
fun rtl (h::t) = t;
```

Firstly, I think it is naive to expect novices to go through this process of making additional definitions. Secondly, it is impossible to define such restricted definitions in *CYNTHIA* because all functions have to be exhaustively defined. Exceptions could be used to give definitions such as `rtl` with an exception in

the base case. However, this still requires an extra definition from the user. I overcome the difficulty by placing side-conditions on reducer lemmas. Hence, under the definition of `tl` given, a reducer lemma would be derived as:

$$\texttt{x} \neq \texttt{nil} \rightarrow \texttt{tl(x)} \leq_w \Pi_2(\texttt{x})$$

where $\Pi_2(x)$ projects x onto its second argument. By embedding the condition implicitly in the parameter to `tl`, this becomes:

$$\texttt{tl}\ (\texttt{h} :: \texttt{t}) \leq_w \texttt{t}$$

- **Extensions to the language.** The language in [McAllester & Arkoudas 96] does not include ML `case` expressions or local function declarations. It does allow local variable declarations but only of the form $dec = exp$ where dec is a variable. dec may be a pattern in ML.

Figure 6.6 gives the inference rules used in Walther Recursion analysis. Recall that the induction needed to prove the lemmas is encapsulated within the IND rule in Figure 6.4. Combined with this induction, the rules in Figure 6.6 can be used to show termination of a wide variety of functions. The rules in fact define the relation $\leq_w$. They work by decomposing the arguments of $\leq_w$ in a way that preserves the measure, w. WCONS1 and WRED apply previously derived RC lemmas or an induction hypothesis. Recall that R_c is the set of recursive arguments of the constructor c. WCONS2 strips off the outermost constructors on each side of $\leq_w$. WCONS3 is similar but for the case when there is not an identical constructor on each side. WCONS4 applies a RC lemma in a different way to WCONS1. WSUBST makes substitutions that have arisen from the rules in Figures 6.4 and 6.5. The following examples makes this clear.

Each time a `let` expression is introduced, *CYNTHIA* stores an equality lemma for each new local variable. For an expression

```
let val z = e in h::z
```

where `z` is a variable and `e` is an expression, *CYNTHIA* stores the lemma `z = e`. Suppose now that we need to prove the goal $\texttt{h} :: \texttt{z} \leq_w \texttt{h} :: \texttt{e}$. WSUBST can be used first to replace `z` by `e` and then WREFL proves the goal. The version of WSUBST in *CYNTHIA* is actually a more general version of the one given here. Consider an expression

$$\frac{}{H \vdash x \leq_w x} \text{ WREFL}$$

$$\frac{H \vdash u_i \leq_w t}{H, (f \ldots x_i \ldots) \leq_w x_i \vdash (f \ldots u_i \ldots) \leq_w t} \text{ WCONS1}$$

$$\frac{H \vdash u_i \leq_w t}{H, (f \ldots c_j(\ldots, x_i, \ldots) \ldots) \leq_w x_i \vdash (f \ldots c_j(\ldots, u_i, \ldots) \ldots) \leq_w t} \text{ WRED}$$

$$\frac{(\forall i \in R_c) \quad H \vdash u_i \leq_w t_i \qquad (\forall i \in \{1, \ldots, n\}) \quad \vdash i \notin R_c \rightarrow (u_i = t_i)}{H \vdash c(u_1, \ldots, u_n) \quad \leq_w \quad c(t_1, \ldots, t_n)} \text{ WCONS2}$$

$$\frac{H \vdash u \leq_w t_i \quad H \vdash i \in R_c}{\vdash u \leq_w c(\ldots, t_i, \ldots)} \text{ WCONS3}$$

$$\frac{H \vdash u \leq_w (f \ldots t \ldots)}{H, (f \ldots x_i \ldots) \leq_w x_i \vdash u \leq_w t} \text{ WCONS4}$$

$$\frac{H \vdash (u \leq_w t) \bullet \{x_2/x_1\}}{H, Y : x_1 = x_2 \vdash u \leq_w t} \text{ WSUBST}$$

Figure 6.6: Rules for Walther Recursion.

```
let val (z1,z2) = (e1,e2)
```

where `z1`, `z2` are variables, *CYNTHIA* stores the lemma `(z1,z2) = (e1,e2)`. WSUBST is extended to recursively extract new equality lemmas from this one — namely, `z1 = e1` and `z2 = e2`.

The `case` construct is also dealt with in *CYNTHIA*. Again, each time a `case` expression is introduced, equality lemmas are stored. As an example, consider the following code:

```
case z of nil => ...
        |   (h::t) => ...
```

where `z` is a variable. This code results in two sub-branches of the proof tree, the first containing the lemma `z = nil` and the second containing the lemma `z = h::t`. WSUBST is used as before.

More generally, `z` does not have to be a variable but may be any ML expression. If `z` is of the form $g(x_1, \ldots, x_n)$ and there exists a RC lemma for g of the form $g(x_1, \ldots, x_n) \leq_w$

x_i then a combination of WCONS4 and WSUBST may be used to obtain a proof. For example, suppose the user has written:

```
case g x of nil => nil
       |    (h::t) => t
```

in the second branch there is an equality lemma `g x = h::t`. Suppose that *CYNTHIA* is trying to prove the bounding lemma $\mathtt{t} \leq_w \mathtt{x}$. If there exists a lemma for `g` of the form $\mathtt{g\ x} \leq_w \mathtt{x}$, applying WCONS4 refines the goal to:

$$\mathtt{t} \leq_w \mathtt{g\ x}$$

Applying WSUBST refines this to:

$$\mathtt{t} \leq_w \mathtt{h :: t}$$

which holds.

Dealing with local function declarations is straightforward. Each local function may have its own RC lemmas (see LET FUN in Figure 6.4) so *CYNTHIA* checks for these as the functions are defined.

To summarise, rules WREFL, WCONS1, WRED, WCONS2 are re-statements of rules in [McAllester & Arkoudas 96]. All other rules are new.

Extended Example

An example of rule application may be illustrative. Consider the `partition` example again. The specification is given in (6.5), p. 121. Proceeding in a backwards fashion, the application of inference rules will refine this specification until all goals are of the form $\vdash \mathit{true}$. All instantiations of variables in the following are given by the user. In practice, an entire proof attempt such as that given here will never be necessary. The theorem proving process always starts with an existing proof and makes modifications as set out in §6.2. The example below is meant to make concrete the description in the previous parts of this chapter.

The first rule to be applied is the I-$\forall$ rule from Chapter 5, p. 92. The subgoal thus

produced is:

$$z_1 : (int * int \rightarrow bool) \vdash P_1 : (\ \forall z_2 : int.\ \forall z_3 : int\ list.$$
$$(f\ z_1\ z_2\ z_3) : int\ list\ \wedge\ (f\ z_1\ z_2\ z_3) \leq_w z_3)$$

where P has been instantiated to $\lambda z_1.P_1$[6]. Two further applications of I-$\forall$ now give the subgoal:

$$z_1 : (int * int \rightarrow bool), z_2 : int, z_3 : int\ list \vdash$$
$$P_2 : ((f\ z_1\ z_2\ z_3) : int\ list\ \wedge\ (f\ z_1\ z_2\ z_3) \leq_w z_3)$$

where P_1 has been instantiated to $\lambda z_2.\lambda z_3.P_2$.

The next rule which is applied is IND. In this example, the form of the IND rule used is as follows:

$$\frac{\begin{array}{l} H \vdash a_{b_1} : ((f\ z_1\ z_2\ \texttt{nil}) : int\ list\ \wedge\ (f\ z_1\ z_2\ \texttt{nil}) \leq_w \texttt{nil}) \\ H, h : int,\ t : int\ list, (f\ z_1\ z_2\ t) : int\ list, X_1 : (f\ z_1\ z_2\ t) \leq_w t \\ \quad \vdash a_{s_1} : ((f\ z_1\ z_2\ (h :: t)) : int\ list\ \wedge\ (f\ z_1\ z_2\ (h :: t)) \leq_w (h :: t)\ \wedge\ t \leq_w t) \end{array}}{z_3 : int\ list\ \vdash\ (ind(z_3, a_{b_1}, \lambda h.\lambda t.\lambda X_1.a_{s_1}(t))) : (f\ z_1\ z_2\ z_3) \wedge (f\ z_1\ z_2\ z_3) \leq_w z_3}$$

This rule mirrors the structure of the patterns in the definition of `partition` — i.e. there is a case for `nil` and a case for `h::t`. It checks that the recursive call is measure decreasing ($t \leq_w t$). It also tries to prove the RC lemma by induction. By applying IND, P_2 is instantiated to:

$$ind(z_3, a_{b_1}, \lambda h.\lambda t.\lambda X_1.a_{s_1}(t))$$

The IND rule gives rise to two subgoals. Consider the base case first:

$$\ldots \vdash a_{b_1} : ((f\ z_1\ z_2\ \texttt{nil}) : int\ list\ \wedge\ (f\ z_1\ z_2\ \texttt{nil}) \leq_w \texttt{nil})$$

The base case continues by applying WITNESS where a_{b_1} is instantiated to `nil`. This instantiation is in general provided by the user and is the one used here because it is the result in the base clause in the definition of `partition`. WITNESS gives us three subgoals:

$$\ldots \vdash \texttt{nil} : int\ list \tag{6.6}$$

[6] I will omit typing information in the extract term for sake of clarity.

$$\ldots \vdash \texttt{nil} \in \Sigma \tag{6.7}$$

$$\ldots \vdash \texttt{nil} \leq_w \texttt{nil} \tag{6.8}$$

All of these subgoals are proved easily. (6.6) is proved by the tactic WFFTACS. (6.7) is proved by the SYNTACS tactic, and (6.8) is proved using WREFL.

The step case subgoal is as follows:

$$H, h : int,\ \ t : int\ list, (f\ z_1\ z_2\ t) : int\ list, X_1 : (f\ z_1\ z_2\ t) \leq_w t$$
$$\vdash a_{s_1} :\ \ ((f\ z_1\ z_2\ (h :: t)) : int\ list\ \wedge\ (f\ z_1\ z_2\ (h :: t)) \leq_w (h :: t)\ \wedge\ t \leq_w t)$$

Instantiating a_{s_1} to $\texttt{if}\ z_1(h, z_2)\ \texttt{then}\ E_2\ \texttt{else}\ E_3$, we can apply IF. This gives four subgoals. Type-checking and semantics-checking are done using WFFTACS and SYNTACS (as before). The other two subgoals correspond to each branch of the conditional split. Let us consider the first branch only. The subgoal in this branch is:

$$\ldots, X : z_1(h, z_2), (f\ z_1\ z_2\ t) : int\ list, X_1 : (f\ z_1\ z_2\ t) \leq_w t$$
$$\vdash E_2 : ((f\ z_1\ z_2\ (h :: t)) : int\ list\ \wedge\ (f\ z_1\ z_2\ (h :: t)) \leq_w (h :: t)\ \wedge\ t \leq_w t)$$

Now we apply WITNESS, instantiating E_2 to $h :: (f\ z_1\ z_2\ t)$. Again, type-checking and semantics-checking are dealt with easily. The remaining subgoal is:

$$\ldots, X : z_1(h, z_2), (f\ z_1\ z_2\ t) : int\ list, X_1 : (f\ z_1\ z_2\ t) \leq_w t$$
$$\vdash (h :: (f\ z_1\ z_2\ t)) : int\ list\ \wedge\ (h :: (f\ z_1\ z_2\ t)) \leq_w (h :: t)\ \wedge\ t \leq_w t$$

There are three conjuncts to prove. The first is trivial. The second needs to be proved using the rules for Walther Recursion and an induction hypothesis. First, apply WCONS2. This gives the subgoal:

$$\ldots \vdash (f\ z_1\ z_2\ t)) \leq_w t$$

which is proved by the induction hypothesis. The third conjunct is easily proved using WREFL.

The second branch of the conditional statement can be proved similarly. Collecting together all the instantiations, P has been instantiated to:

$$\lambda z_1.\lambda z_2.\lambda z_3.ind(z_3, \texttt{nil}, \lambda h.\lambda t.\lambda X_1.\texttt{if}\ z_1(h, z_2)\ \texttt{then}\ h :: (f\ z_1\ z_2\ t)\ \texttt{else}\ (f\ z_1\ z_2\ t))$$

A simple translation, along with a mechanism for keeping track of variable names, gives the program `partition`.

6.2 Replaying Proofs According to User Edits

When the user applies an editing command to the current program, *CYNTHIA* must apply a corresponding edit to the current synthesis proof. Typically, this edit will make an isolated change to the proof. *CYNTHIA*'s analogical mechanism then propagates this change through to the rest of the proof. This section describes this analogical mechanism in detail.

Abstract Rule Trees

As mentioned earlier in this chapter, analogy is more successful the higher the level of abstraction. The right level of abstraction here is that of tactic applications. Following [Madden 91], I make the following definition.

Definition 7 *Let a synthesis (refinement) proof be represented as a tree of nodes where the nodes are of the form $H \vdash G$ for hypotheses H and goal G and the arcs joining nodes are rule or tactic applications. Then the Abstract Rule Tree (or ART) of this proof is precisely the sequence of tactic*[7] *applications connected by the tactical* ' then'.

Hence, the ART abstracts away information about the hypotheses and goals. Note that since ARTs are tactic applications, they can be applied to reform the synthesis proof from which they came. The procedure for editing the proof is as follows. The user highlights the position in the program where he wishes to make a change. *CYNTHIA* calculates the corresponding position, *pos*, in the proof tree. Let the synthesis proof be denoted by P_t and the proof subtree below *pos* by P_s. *CYNTHIA* abstracts P_s into an ART A_s. *CYNTHIA* then makes changes to A_s to give $\phi(A_s)$. $\phi(A_s)$ is then unabstracted or replayed to give the new proof subtree $\phi(P_s)$. The complete new proof tree is then P_t with P_s replaced by $\phi(P_s)$. Note that *CYNTHIA* abstracts only P_s and not the whole proof tree P_t. This saves effort because, due to the refinement nature of the proofs, any tactics not in P_s will be unaffected.

Some commands also require a change to the specification. For example, ADD CURRIED ARGUMENT adds an additional type to the specification. Changes to the specification

[7] Again, I shall use 'tactic' in situations where either 'tactic' or 'rule' would be applicable.

are easy to make.

Note that all changes made by *CYNTHIA* are made to the underlying proof representation, not to the actual program. The new program is obtained by translating and re-displaying the proof rather than just manipulating the existing program display.

Selective Replay of Tactics

Correctness-checking tactics (type (3)) can be time-consuming and so *CYNTHIA* selectively replays these tactics. *CYNTHIA* automatically decides which correctness-checking tactics need to be replayed according to which editing command was applied. Type (1) and type (2) tactics are always replayed.

Consider type-checking tactics. In some cases, expressions within the ML program will not need to be type-checked during the replay. Consider applying the ADD CONSTRUCT command to introduce a conditional `if then else` statement into the program. This will copy the highlighted expression, `E`, to each branch of the condition to give:

```
if C then E else E
```

where `E` has been copied. Clearly, there is no point type-checking `E` during the replay as its status will be unchanged. Some commands will require that `E` is type-checked, however. If CHANGE TYPE is used to change the top-level signature, then the target synthesis proof may require `E` to inhabit some new type. We must apply type-checking to see if this holds.

There turn out to be three possibilities, corresponding to the second column in Table 6.1.

- Replay **all** type-checking in P_t. Changing type, say from t_o to t_n, means that some expressions in the source proof of type t_o may become ill-typed. Hence, when invoking CHANGE TYPE all type-checking subgoals are processed. Some of these type subgoals will be satisfied and some will no longer hold and so will be fed back to the user as type errors.

- Replay type-checking at the **current** node only. Some commands only require type-checking subgoals to be proved at the current node in the proof tree. For

example, if a new witness is introduced by CHANGE TERM, we need only check that the new witness is of the correct type. Expressions in other parts of the proof tree will be unchanged and hence will still be of the correct type. Hence, in the **current** case, *CYNTHIA* invokes type-checking at the current node, but not at other nodes. At other nodes, type-checking obligations are just copied from the source proof rather than being replayed. Hence, if a source obligation was (un)satisfied in the source, so will it be in the target.

- Replay **none** of the type-checking subproofs. Commands such as ADD ARGUMENT do not affect type obligations in the source proof — if the user adds a new input $y : T$ then the top-level type is automatically updated to include T. Since no expression in the source proof contains y, the only expressions that will become ill-typed under ADD ARGUMENT are recursive calls (because the source recursive calls now have an input missing). However, all recursive calls are automatically given an additional input by the command anyway, so the target recursive calls are necessarily well-typed and so need not be checked. Any type-checking subgoals will hold if they held in the source and will not hold if they were false in the source. In this case, source type-checking information is just copied to the target.

Command	Types	Termination
CHANGE TERM	**current**	**both**
ADD RECURSIVE CALL	**current**	**measure**
REMOVE RECURSIVE CALL	**none**	**none**
MAKE PATTERN	**none**	**none**
REMOVE PATTERN	**none**	**none**
ADD CONSTRUCT	**none**	**none**
REMOVE CONSTRUCT	**none**	**none**
CHANGE TYPE	**all**	**none**
ADD ARGUMENT	**none**	**none**
REMOVE ARGUMENT	**none**	**none**
RENAME	**none**	**none**

Table 6.1: Editing Commands and How They Affect the Source Replay.

Another time-consuming aspect of the replay of the source proof is the termination analysis. It can be noted, however, that most editing commands (see column 3 of Table 6.1) will not affect termination of the function being defined. In fact, there are only two situations in which termination analysis needs to be carried out. In all other

cases, it is enough just to copy across the source subproofs that deal with Walther Recursion without actually applying the rules in these subproofs.

- Changing a witness node. If a witness is changed (by CHANGE TERM) then any bounding lemmas that may have been derived for the current function may no longer hold. Hence, reducer / conserver lemma analysis needs to be carried out in the target. Measure argument analysis must also be done in the target even though no new recursive calls are introduced. This is because functions involving nested recursive calls may use their own reducer / conserver lemmas in the measure argument analysis. For example, consider the definition:

```
fun f nil = [ nil ]
|   f (h::t) = h:: f (f t);
```

  Proving termination of this function requires the conserver lemma `f z` $\leq_w$ `z` which is true. However, if CHANGE TERM is used to modify `nil` to `0::nil` at the indicated point, the lemma no longer holds and the function no longer terminates. Hence, measure argument analysis must be carried out in the target — this analysis would alert the user to non-termination. Note that if the recursive calls were not nested, the measure argument analysis would still hold as it would not involve lemmas about `f` itself.

- Changing the recursive calls. If a new recursive call is introduced, it needs to be checked to be measure-decreasing. Hence, the measure argument analysis component of Walther recursion is carried out in the target. Note that the RC lemma analysis need not be done because ADD RECURSIVE CALL only makes the new recursive call available for use, it does not actually introduce it into the program. This is done by a call of CHANGE TERM.

The only command that can affect the truth of the RC lemmas is CHANGE TERM. Hence, whenever CHANGE TERM is applied, WR analysis should be carried out. In fact, it is not necessary to carry out the entire analysis as some of it can be re-used. The current implementation of *CYNTHIA* always carries out measure argument analysis, but will attempt to retain parts of the proof corresponding to RC analysis. For conserver lemmas, the procedure adopted is as follows. First make the following definition.

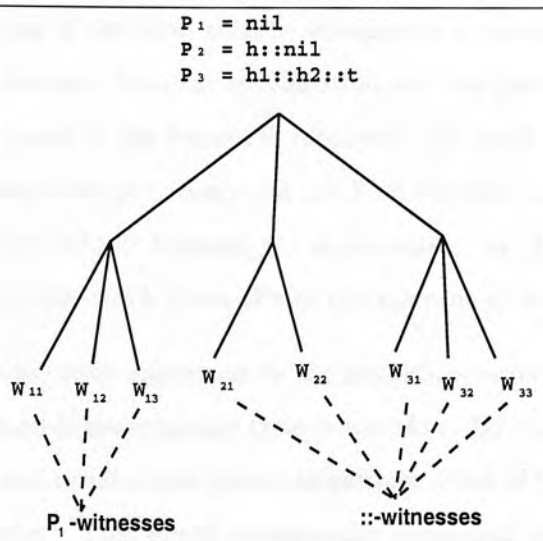


Figure 6.7: P_i- and c_j-witnesses.

Definition 8 *Suppose a function f is defined by the set of well-defined patterns $\{P_1, \ldots, P_k\}$ implemented by an* IND *rule. Then the set of P_i-witnesses is the set of witnesses lying below the* IND *rule in the branch for the pattern P_i. The set of c_i-witnesses is $P_{i_1} \cup \ldots \cup P_{i_r}$ where P_{i_j}, $1 \leq j \leq r$, has top-level constructor c_i.*

See Figure 6.7 for an example. To prove a conserver lemma, $(f\ x_1 \ldots x_n) \leq_w x_{i_j}$, it needs to be shown that the lemma holds at all witness nodes. To prove a reducer lemma, $(f\ x_1\ \ldots c_j(\ldots, r_{jk}, \ldots) \ldots\ x_n) \leq_w r_{jk}$ where c_j is a constructor, the lemma must hold for all witnesses in the set of c_j-witnesses.

Suppose the user applies CHANGE TERM to the witness W_{kl}. Then (referring to Definition (6.4)):

1. For each $j \in \{i_1, \ldots, i_r\}$:
   Check if $W_{kl} = (f\ x_1 \ldots x_n) \leq_w x_{i_j}$ still holds. If so, then this RC lemma remains in the specification. Otherwise, remove it from the specification and update the rest of the synthesis proof.

2. For each $j \in \{1, \ldots, n\}$ such that $j \notin \{i_1, \ldots, i_r\}$:
   $W_{kl} = (f\ x_1 \ldots x_n) \leq_w x_{i_j}$ is a potential new RC lemma. Try to prove this lemma. If it can proved, add it to the specification.

Basically, (1) says that if the new witness invalidates a particular conserver lemma then that lemma is removed from the specification and any part of the synthesis proof that deals with the proof of the lemma is removed. (2) looks to see if there are any other conserver lemmas that previously did not hold but that now hold because of the change of witness. For reducer lemmas, (1) is the same. In (2) rather than checking all other Ws we need only check those Ws in the relevant c_j-witness.

These optimizations are very important to the smooth operation of *CYNTHIA*. The system will not be used if the response time is too slow. By eliminating the replay of some type-checking and termination proof obligations, some of the most time-intensive tasks have been avoided[8]. This would be especially important in possible future extensions of *CYNTHIA*. At present, the replay is done a definition at a time. A natural extension would be to allow the user to make a change in one definition and have this change propagated through other, dependent definitions. This would involve the replay of multiple synthesis proofs for a single edit. In such circumstances, further optimizations would be needed to ensure a decent performance.

Failed Proof Obligations

Normally, a proof is thought of as a tree of inference rule applications and since rules are sound, the proof cannot contain errors. However, it was described in Chapter 4 how the structure of the editing commands make it impossible to retain correct programs at all times. Incorrect programs are represented as incomplete proofs. During the analogical replay of the source rules, if a rule no longer holds its application will fail in the target proof. In some cases, the rule can be modified so that it goes through. If this cannot be done, however, a gap is left in the target proof. This corresponds to a position in the ML program and so the program fragment corresponding to where the proof failed can be highlighted to the user. This failed proof rule usually denotes a type error or another kind of static semantic error such as an unbound variable.

As an example, consider the following program which contains type errors:

```
int -> int list -> int
```

[8] The same techniques could be used to selectively replay SYNTACS but in practice this tactic is very efficient so the gains would be only minor.

```
fun delete e nil = [ nil ]
|  delete e (h::t) = if e=h then delete e t
                     else [ h :: ] delete e t;
```

The boxes denote locations whose types are inconsistent with respect to the declared result type of the function. Figure 6.8 gives an abstract representation of the synthesis proof for the program. Recall that triangular boxes denote correctness checks and rectangles represent tactic applications. The dotted boxes denote failed proof obligations corresponding to the two type errors in the program. Note how the failed proof obligations correspond to specific locations in the program. *CYNTHIA* works by highlighting the parts of the program that correspond to failed obligations (precisely the boxes in this instance).

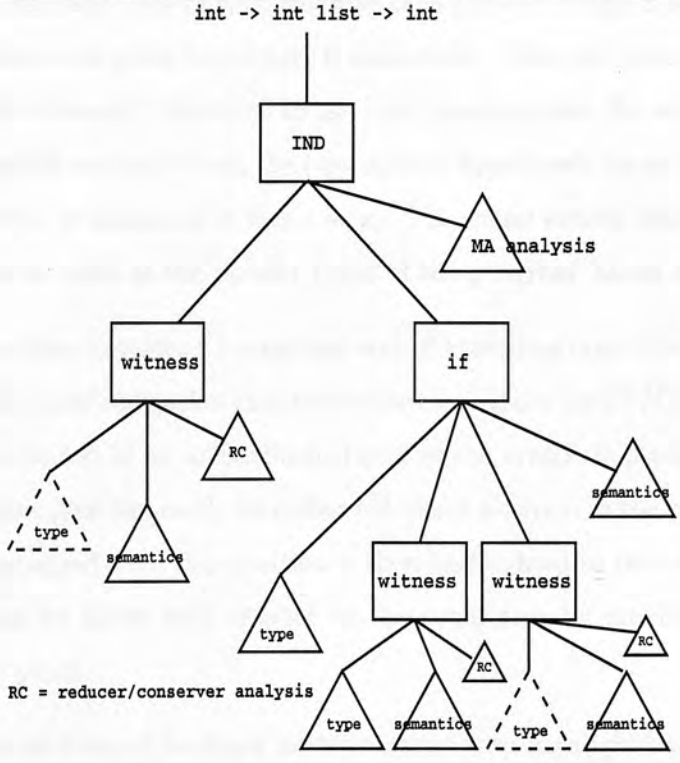


Figure 6.8: A Synthesis Proof for a Buggy Version of `delete`.

6.3 Hiding the Proof from the User

CYNTHIA is intended to be used by novices. Hence, it is imperative to package the synthesis proof in such a way that the user is unaware of its existence. The user should be concerned only with the editing commands that he should apply to succeed. He

should not be concerned with which tactics or rules he needs to apply at a certain point. In general, presenting proofs in a clear and concise manner is a difficult problem — cf. [Huang 94]. However, *CYNTHIA* proofs are very simple in nature. The use of tactics that correspond closely to ML programming constructs greatly simplifies the translation of synthesis proofs to ML programs. The translation essentially involves abstracting away proof-checking obligations including type-checking and termination-checking, plus a syntactic translation to convert a proof tree into a code fragment.

Editing commands will have a different effect depending on the position within the proof tree at which they are applied. For this reason, it is necessary to keep track of which program fragments correspond to which parts of the synthesis proof tree. A simple mechanism takes care of this, whereby each piece of program text is tagged with the position within the proof tree where it came from. This also means that contextual information can be readily displayed to the user upon request. For example, if the user clicks on the middle mouse button, the (translated) hypothesis list at the corresponding point in the proof is displayed in menu form. The menu entries state precisely which expressions can be used at the present point of the program, along with their types.

This mechanism also provides a convenient way of providing error feedback information to the user. Any proof obligation that cannot be established by *CYNTHIA*'s analogical mechanism will be left as an unestablished goal in the synthesis proof. The position of this unestablished goal can easily be converted into a position in the program. The text of the program tagged with this position is then highlighted to the user. Explanations of the error can be given with respect to the proof tree by moving to the relevant position in the proof.

The most common form of feedback in this manner is to highlight bad static semantics or ill-typed expressions. However, it may also be helpful for alerting the user to non-terminating programs. In general, *CYNTHIA* will disallow non-terminating programs so that the user will not be able to produce them. The way this works is that if the user enters (using ADD RECURSIVE CALL) a new recursive call for a function, and if the call cannot be shown to be measure-decreasing then the user will just be told that his command cannot be accepted because termination could not then be proved. However, it is possible to get around this restriction by applying certain sequences of edits.

Consider the following function definition:

```
fun ssort nil = nil
|   ssort (h::t) = let fun replace x y nil = nil
                       |   replace x y (h::t) =
                              let val z = replace x y t
                               in if h=x then y::z else h::z
                              end
                   in
              min (h::t) :: ssort(replace (min (h::t)) h t)
                   end;
```

A conserver lemma can be derived for `replace`, namely `replace x y t` $\leq_w$ `t`. This allows the system to derive termination of `ssort`. Suppose, however, that the user now invokes CHANGE TERM to change `nil` in the base case of `replace` to `[x]`, say. The conserver lemma for `replace` is no longer valid and so `ssort` can no longer be proved terminating. This is problematic because the (now) non-measure-decreasing recursive call for `ssort` is still in the definition. The user can be notified of this fact by highlighting the recursive call for `ssort`. The failed proof obligation that will cause this highlighting is the MA obligation produced by the IND rule associated with `ssort`.

The tagging of positions can be extended to positions within expressions. One aspect of *CYNTHIA*'s interface is that if the mouse is placed over a function symbol, the sub-expression with that function symbol as top-level operator is highlighted. When an expression is printed to the display, each sub-expression within an expression is tagged with its position within the surrounding expression. Hence, when the mouse is placed over an expression, sub-expressions with the same position tag are highlighted. If an editing command is applied to a sub-expression, the position tag is used to edit the correct part of the proof tree.

6.4 Summary

This chapter has described how ML function definitions may be encoded as synthesis proofs by creating tactics that correspond to ML constructs and ML programming concepts. Additional tactics ensure the correctness of the ML definition. When an editing command is applied by the user, *CYNTHIA*'s analogical mechanism is used to update the synthesis proof by propagating the effects of the edit throughout the proof.

The technique known as Walther Recursion has been incorporated into the synthesis proof and it is this technique that guarantees the termination of programs defined in *CYNTHIA*.

"OK. Now I'll show you how to navigate through this forest." Cynthia had led Mavis for the past two and a half hours. Mavis was getting tired and was actually beginning to doubt Cynthia's claims. At last, she thought to herself. Maybe Cynthia's finally going to show me the way to the castle.

The pair had come to yet another busy junction. Once again, each path leading away had a cryptic message written on a sign above it. Mavis had long since given up trying to make any sense out of them. Glancing around, Mavis could see that the signs said things like "This path leads to water", "You'll need to fly through here", "No clues down this path", "Just give up now". Well, in truth, this last sign didn't really say that. But it might as well have done.

Cynthia directed Mavis towards one side of the junction. "Wait here," she commanded. Cynthia then moved back towards the centre of the crossroads and started taking out various ingredients from her pouch.

"What are you doing?" asked Mavis.

"You'll see," came the reply. Cynthia had her back to Mavis so she couldn't quite make out what was going on. But Mavis could tell that Cynthia was making some sort of potion. Once this was finished, Cynthia gathered some twigs that happened to be lying around and, with a piece of flint, she lit a small fire. Cynthia then started sprinkling herbs over the fire. Suddenly, the smoke from the fire turned a bright pink colour. Seconds later, the fire started glowing incredibly strongly and then was a large bang. Mavis cowered in fear.

When Mavis lifted her head up, there was smoke everywhere. "Where are you?" said Mavis.

"Don't worry," answered Cynthia. And as the smoke began to clear, Mavis could do nothing but sit back in awe and amazement. All the signs to all the paths had changed. Most of them now had one word written on them in very bold letters: "WRONG". And one of the paths had words that were music to Mavis's ears. "To the castle," it said.

Chapter 7

Mechanisms Underlying the Editing Commands

Chapter 4 described the editing commands from the perspective of a *CYNTHIA* user. This chapter describes the mechanisms underlying the more complex editing commands. The other commands fit easily into the proof framework in Chapter 6. However, some commands require additional machinery — notably CHANGE TYPE, MAKE PATTERN and REMOVE PATTERN. This machinery is explained in this chapter.

7.1 Editing Command Framework

The editing commands inhabit a framework that has been designed in such a way as to make adding, removing or modifying editing commands as easy as possible. Figure 7.1 is a template describing the representation of editing commands within *CYNTHIA*. The preconditions must hold before the command can be applied. `Spec_changes` describes modifications to the specification and `Proof_changes` to the abstract rule tree (ART)[1]. `Termination` modifies the ART to include proofs of RC lemmas or measure argument analysis. Some editing commands require termination analysis to be redone and for some commands the proofs of termination are unaffected. The entries in `Termination` describe which parts of termination analysis are replayed. A breakdown of this is given in Table 6.1, p. 134.

[1] Recall from Chapter 6 that the ART is a tree of rule applications. It is an abstraction of the proof tree, where the goals and hypotheses at each node have been abstracted away.

```
command(Name(...Args...)
        Preconditions
        Spec_changes
        Proof_changes
        Termination
       )
```

Figure 7.1: The Editing Command Structure.

When an editing command is applied, it is applied directly to the ART, as described in Chapter 6. The following steps take place when an edit is applied:

1. **Validate command**. The parameters of the command are checked for errors.
2. **Copy the proof**. The entire proof tree is copied into a store so that if the user goes on to invoke UNDO, the previous proof can be retrieved easily. The UNDO command only allows the most recent edit to be undone.
3. **Store position**. Some commands require the current position within the proof tree to be changed. An example, is RENAME which must process the entire proof tree to ensure that all variable occurrences are changed. Hence, RENAME initially moves the current position to the top of the tree. To ensure that the position is the same once the edit has been applied, the original position is stored.
4. **Access command**. The template representation of the command is retrieved and the actual parameters are substituted for generic parameters throughout.
5. **Check preconditions** (`Preconditions` in Figure 7.1).
6. **Abstract rule tree**. This abstracts the proof tree to an ART.
7. **Execute changes to specification** (`Spec_changes` in Figure 7.1).
8. **Execute changes to the ART** (`Proof_changes` in Figure 7.1).
9. **Unpack ART**. From the current position, the proof rules in the ART are replayed.
10. **Execute termination analysis** if necessary (`Termination` in Figure 7.1).
11. **Return position**

Once this process is finished, a new proof tree is obtained and the corresponding program is extracted and displayed to the user.

7.2 Changing Type

The most complex editing command is that for changing the type of an expression. The complexity of the CHANGE TYPE command varies according to which argument it is applied to. There are two cases to consider, based on the following definition.

Definition 9 *Suppose a function f has arguments $x_1, \ldots, x_n$ where these arguments may be curried or uncurried. Then x_i is said to be a* split argument *if f is defined by pattern matching over x_i. Otherwise, x_i is an* unsplit argument.

Example

In the program:

```
fun combine f (nil, nil) = nil
|   combine f (nil, y::ys)   = raise excep
|   combine f (x::xs, nil)   = raise excep
|   combine f (x::xs, y::ys) = f(x,y) :: combine f (xs,ys);
```

`combine` has one unsplit argument, `f`, and two split arguments. □

The behaviour of CHANGE TYPE depends on whether the selected argument is split or unsplit. In both cases, `Spec_changes` (recall Figure 7.1) updates the specification to reflect the type change. If the argument is unsplit, then `Proof_changes` is empty. The source proof tactics are replayed as they were in the source (see Chapter 6). Of course, the change of type may have invalidated part of the proof. This corresponds to the failure of type-checking proof obligations which are relayed to the user as type errors. If the argument is split, then the patterns that the argument has been split into must be changed also. These changes must be in such a way that the resulting pattern definition of the function is well-defined. In the case of split arguments, `Proof_changes` is non-empty. In this case, CHANGE TYPE modifies a set of patterns which may involve adding (or removing) clauses to the function definition. In proof terms, the IND rule is modified so that the target IND node has a different number of subgoals than the

source. New proof branches correspond to new program clauses. This results in the following steps being carried out:

1. Find a relation between the old and new datatype.
2. Update the relevant source IND rule according to this relation.
3. Add or remove branches in the proof tree below the IND tactic node, depending on whether clauses have been added or removed in the pattern definition.

The CHANGE TYPE command allows the user to make a very broad range of changes to types and *CYNTHIA* will automatically make any necessary modifications to the patterns. A specification of this algorithm is described in this section, and is presented along with a proof that the patterns produced by CHANGE TYPE are well-defined.

7.2.1 Examples

The aim of the CHANGE TYPE algorithm is to transform a set of well-defined patterns to a target set of well-defined patterns, with respect to the change of type. To motivate what follows, let us first consider an example. Suppose the user is about to edit the following `member` function[2]:

```
nat -> [ nat list ]  -> bool
fun member x nil = false                                    (†)
|   member x (h::t) = nateq(x,h) orelse member x t;
```

It may be that the user now wishes to produce a member function for natural number trees and so applies CHANGE TYPE at the indicated point. Note that the function is defined by pattern matching and so the patterns must also be changed:

```
nat -> nat tree -> bool
fun member x (leaf n) = false
|   member x (node(h,t,t2)) = nateq(x,h) orelse member x t;
```

There are two points to note here. First, *CYNTHIA* has automatically introduced new variables, `n` and `t2`. This is necessary because the arity of the constructors are different in the source and target. Second, it may be that the user also wishes to modify

[2] where `nateq` compares two natural numbers for equality

the recursion scheme within the program. `node` has two recursive arguments and so *CYNTHIA* should make available a recursive call for the new recursive argument, `t2`. Recall from Chapter 5 that the user is provided with a list of valid recursive calls. When CHANGE TYPE is applied, `member x t2` will be added to this list. *CYNTHIA* knows that any definition using this new recursive call will still terminate (since `t2` has measure strictly less than `node(h,t,t2)`) and so it is allowed to be introduced.

The CHANGE TYPE command makes all of the above mentioned changes automatically. To do this, requires an algorithm for comparing the datatype definitions of the old and new ML datatype definitions. A relation is derived that associates source and target type constructors. This relation is used to modify the old pattern definition as required. In the above example, the relation would associate `nil` with `leaf`, and `::` with `node`. There is also a relation between the arguments of each constructor pair.

I restrict to a subset of the ML types[3]. Even so, there are a number of significant problems to be overcome to realise this algorithm. First, CHANGE TYPE could be applied in the presence of non-standard patterns. Suppose the user had applied MAKE PATTERN to `t` in `member` († on p. 146):

```
nat -> nat list -> bool
fun member x nil = false                                          (††)
|   member x (h::nil) = nateq(x,h) orelse member x nil
|   member x (h::h2::t) = nateq(x,h) orelse member x (h2::t);
```

The CHANGE TYPE algorithm must be defined in such a way that the target pattern is now:

```
nat -> nat tree -> bool
fun member x (leaf n) = ...
|   member x (node(h,leaf n,t2)) = ...
|   member x (node(h,node(h2,t,t3),t4)) = ...
```

where the change has been applied recursively. Further complications arise, if MAKE PATTERN has been applied on more than one occasion, but to arguments of different types. Consider what happens, if MAKE PATTERN is applied to `h` instead of `t` in (†). This would give:

```
nat -> nat list -> bool
```

[3] specified in §7.2.5, p. 166

```
fun member x nil = ...
|   member x (zero::t) = ...
|   member x (s n :: t) = ...
```

Suppose now that the user wishes to change the type of `nat list` to `nat list tree`. In order to maintain a well-typed program, *CYNTHIA* must derive two relations: as described in Figure 7.2. The principal constructor of the source type is `list` and of the target type is `tree`. First, a relation must be found between these types. Second, a relation must be found between `nat` and `nat list` because these are the types of corresponding arguments in the first relation. In Figure 7.2, arrows connecting ellipses denote relations between types, other arrows denote relations between constructors.

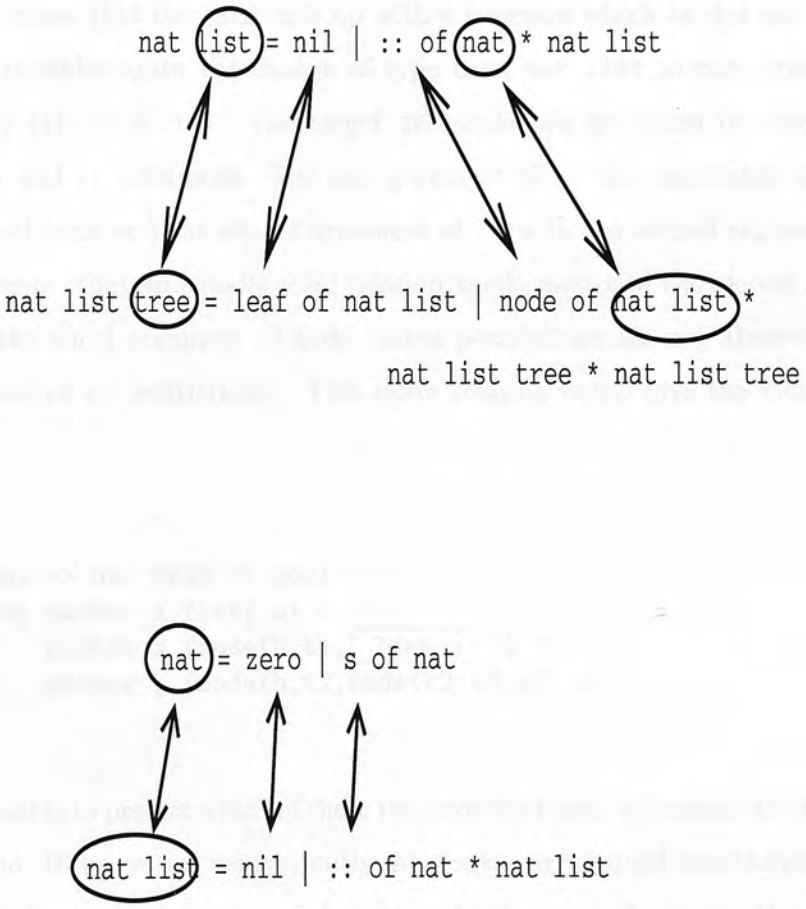


Figure 7.2: Changing type from `nat list` to `nat list tree`.

The resulting program would be:

```
nat -> nat list tree -> bool
fun member x (leaf n) = ...
|   member x (node(nil,t,t2)) = ...
```

```
|   member x (node(n::t3,t,t4)) = ...
```

The flexibility to deal with types such as the above means that CHANGE TYPE must recursively derive relations which are then applied to independent parts of the program.

In general, there will be many possible relations between any two given datatypes. *CYNTHIA* restricts these possibilities in two ways: restrict the relations to those that produce valid constructor terms, and use heuristics to decide between those that remain. Often, this will not limit the choice to a unique relation. In such circumstances, *CYNTHIA* will arbitrarily choose a relation from the restricted set of possibilities. This may mean that the user ends up with a program which he did not intend. For example, consider again the change of type from `nat list` to `nat tree` in `member` marked by (††) on p. 147. The target definition was produced by associating `nil` with `leaf` and `::` with `node`. The first argument of `::` was associated with the first argument of `node` and the second argument of `::` with the second argument of `node`. Note, however, that an equally valid relation would associate the second argument of `::` with the third argument of `node` (other possibilities are not allowed — see the next subsection on restrictions). This latter relation would give the following target program:

```
nat -> nat tree -> bool
fun member x (leaf n) = ...
|   member x (node(h,t1,[ leaf n ])) = ...
|   member x (node(h,t2,node(h2,t3,t)) = ...
```

It is impossible to predict which of these two possible target definitions the user actually has in mind. If the system inadvertently chooses the unintended one, the user can easily apply edits to get to the preferred definition. In this example, he would need to apply REMOVE PATTERN to the indicated occurrence of `leaf n` and then MAKE PATTERN to `t1`. A better solution would be to allow the user to summon up the other possibilities and if necessary, choose an alternative. Although the CHANGE TYPE algorithm does collect all possible relations, the interface has not yet been extended to allow the user to access these alternatives.

7.2.2 Specification of the Algorithm

The strategy embodied in the CHANGE TYPE command is as follows. Recursively find relations between the old and new datatype — at each stage of the recursion, this involves finding a relation between the constructors of the two datatypes and then another relation between the argument positions of each constructor pair. This allows us to associate, with each source constructor term, $c(x_1, \ldots, x_n)$, a (possibly empty) set of target constructor terms. If $d(y_1, \ldots, y_m)$ resides in this set, then the notation $c(x_1, \ldots, x_n) \mapsto d(y_1, \ldots, y_m)$ will be used[4]. A number of restrictions on these two relations are insisted upon. These restrictions are given below, along with a set of heuristics for deciding between possible relations.

Restrictions for the relation between constructor arguments

- If x_i is a recursive argument, and if $x_i \mapsto y_j$, then y_j should also be a recursive argument. This is necessary because any recursive call should recurse on recursive arguments (i.e. arguments that are of a recursive type). Consider an example:

```
'a list -> int
fun length nil = 0
|   length (h::t) = 1 + length t;
```

  Suppose a change of type is specified from `'a list` to `nat` and that the restriction on recursive arguments is not adhered to — i.e. $\texttt{::}_1 \mapsto \texttt{s}_1$, then the resulting program is:

```
nat -> int
fun length zero = 0
|   length (s h) = 1 + length t;
```

  This definition is ill-formed. The recursive call should recurse on the parameter of `s`. Respecting the restriction, $\texttt{::}_1 \mapsto \texttt{s}_1$ is not allowed, but $\texttt{::}_2 \mapsto \texttt{s}_1$ is used instead, giving the program:

```
nat -> int
fun length zero = 0
|   length (s t) = 1 + length t;
```

[4] This notation is also used for the constructor-to-constructor relation. In addition, for constructors c and d, $c_i \mapsto d_j$ means that the ith parameter of c is related to the jth parameter of d.

- Similarly, if x_i is a non-recursive argument such that $x_i \mapsto y_j$, then y_j should also be non-recursive.

- A final restriction is that duplication of arguments is not allowed — e.g. `(h::t)`$\mapsto$ `node(h,t,t)` is disallowed because `node(h,t,t)` cannot be used to define a function using pattern matching. Instead, a fresh variable is introduced, `(h::t)`$\mapsto$ `node(h,t,t2)`.

These constraints can always be satisfied. In the following algorithm specification, §7.2.3, these restrictions are captured by the relation θ_{C,T_1,T_2} which associates constructors (and their arguments) of type T_1 with constructors (and their arguments) of type T_2.

Heuristics for constructor arguments

- If the number of recursive arguments in the source is less than in the target, fresh variables are introduced into the target constructor term. This applies to non-recursive arguments also. A heuristic says that the relation with the fewest number of fresh variables should be chosen. This means that all source arguments must appear in the target term *if possible.* In the case that the number of source (non-)recursive arguments is greater than in the target, then it is impossible for all source arguments to appear in the target. To illustrate, this would mean that `(s n)` $\mapsto$ `e1::n`, where `e1` is a fresh variable, is chosen over `(s n)` $\mapsto$ `e1::e2`, but `(h::t)`$\mapsto$`s t` is allowed, where `h` is "lost". Note that losing arguments in this way may lead to ill-formed target programs if a lost variable `h` is referred to elsewhere in the definition. In this case, the ill-formed expression is highlighted and it is left to the user to change it.

- For non-recursive arguments, an additional heuristic has been implemented. This chooses relations that associate arguments of the same type. Hence, if two constructors were defined as:

```
c : int * nat * 'a list -> 'a list
d : nat * int * 'a tree -> 'a tree
```

  then the system would produce `c(h1,h2,t)` $\mapsto$ `d(h2,h1,t)`. The motivation behind this is that by respecting types, it is less likely that other parts of the

program will become ill-typed. For instance, the source definition may contain a conditional statement on whether the first argument of `h1` is equal to `0`. If the two arguments of `c` are swapped, then this statement remains well-typed.

The restrictions and heuristics are similar when relating constructor symbols rather than constructor arguments.

Restrictions for constructor symbols

- Base (step) constructors relate to base (step) constructors.
- Because the number of source and target constructor symbols may differ, the relation may be 1-1, when the number of source and target constructors is the same, or 1-many, when there are more target than source constructors. If there are more source than target constructors, some source constructors will not appear in the relation — these source constructors are said to be "lost". When there is a 1-many, say $1-n$, relation then the branch of the proof tree corresponding to the source constructor is copied n times in the target. See the example below for clarification. Similarly, in the case that a constructor symbol is "lost", branches of the proof tree are deleted.
- All target constructors must have a pre-image in the source. Otherwise, ill-defined patterns would be produced. (Note that not all source constructors need have an image — since some source constructors may be "lost").

These restrictions are captured by θ_B and θ_S in the following specification (p. 156). There is one instance where the first and second constraints cannot be satisfied. This is when there are no source step constructors but there are some target step constructors. This means that the target step constructors cannot have pre-images that are also step constructors. Since the patterns produced by CHANGE TYPE must be well-defined, the target constructors without pre-images must be introduced explicitly. The parameters of these constructors are fresh variables and a gap is left on the right-hand side of the equation. Consider the following example[5]:

[5] `bool` is a primitive ML type with base constructors `true` and `false`.

```
bool -> bool
fun negate true = false
|   negate false = true;
```

Suppose the user changes the input type to `nat`. The base constructors of `bool` and `nat` are related such that `true`↦ `zero` but there are no step constructors in the source. Hence the second clause below is introduced explicitly by *CYNTHIA*.

```
nat -> bool
fun negate zero = false
|   negate (s n) = ??? ;
```

Heuristics for constructor symbols

The heuristics are different when mapping base constructors and step constructors. For step constructors, the heuristics are:

1. Relate constructors of the same name if possible. If two datatypes are defined (in the ML-style) as:

   ```
   to = ATOM of nat | OR of to * to | AND of to * to;
   tn = ATOM of nat | OR of tn * tn | AND of tn * tn | NEG of tn;
   ```

   Then `OR` ↦ `OR` is made in preference to `OR` ↦ `AND`.

2. Preserve the type of constructors in the relation (up to renaming of `to`, `tn`). As an example, consider:

   ```
   to = ATOM of nat | OR of to * to | NEG of to;
   tn = ATOM of nat | NEG of tn | AND of tn * tn;
   ```

   `OR` ↦ `AND` is chosen over `NEG` ↦ `AND` because `OR`, `AND` have the same type (up to renaming of `to`).

3. Preserve the number of recursive arguments. This is a relaxed form of the previous heuristic. Given a constructor, `Cons` with argument type `nat * tn` then `NEG` ↦ `Cons` is chosen over `OR` ↦ `Cons` because `NEG`, `Cons` both have a single recursive argument.

In the case of mapping base constructors, heuristics (1) and (2) still apply. Heuristic (3) does not because there are no recursive arguments.

Example

Suppose `to`, `tn` are defined as:

```
to = ATOM of nat | NEG of to | AND of to * to;
tn = ATOM of nat | OR of tn * tn | LABEL of tn * nat | NEG of tn;
```

Consider the function definition[6]:

```
fun f (ATOM n) = n
|   f (NEG p) = (f p)
|   f (AND(p, q)) = plus ((f p), (f q));
```

Then the relation between constructors is given by `ATOM` $\mapsto$ `ATOM`, `NEG` $\mapsto$ `NEG`, `AND` $\mapsto$ `OR`, `NEG` $\mapsto$ `LABEL` giving a new function:

```
fun f (ATOM n) = n
|   f (NEG p) = (f p)
|   f (LABEL(p, e1)) = (f p)
|   f (OR(p, q)) = plus((f p), (f q));
```

Note how the 1-many relation of `NEG` has increased the number of cases in the pattern definition. As a further illustration, consider the reverse: sending `tn` to `to`. In this case the relation is given by `ATOM` $\mapsto$ `ATOM`, `NEG` $\mapsto$ `NEG`, `OR` $\mapsto$ `AND` and `LABEL` is "lost". The new function has no pattern for `LABEL`. □

By heuristically choosing a relation, it is possible that we could confuse/misdirect the user. For instance, suppose the user actually wanted to lose the case for `OR` rather than `LABEL` above. This is an unusual situation, however, as *CYNTHIA* is intended to be used in producing complex programs from simple ones rather than the other way round. So it is less likely that branches of the program would be lost during editing. In any case, the user can obtain the intended program by applying further editing commands.

7.2.3 Specification of CHANGE TYPE

This section gives a detailed specification of the CHANGE TYPE editing command. It also serves as necessary technical machinery for showing the correctness of the command. Correctness in this context means that applying the command to a well-defined

[6] where `plus` is addition over the natural numbers

set of patterns will always produce a well-defined set of patterns (where well-defined is in the sense defined in Chapter 4 (p. 65) and repeated in Definition 11).

Definition 10 *An expression* $c(x_1, \ldots, x_n)$, *where* n *may be* 0, *is said to be a* c-term *if* c *is a constructor and for* $1 \leq i \leq n$, x_i *is a distinct variable. The expression is a* constructor term *if each* x_i *is either a variable or a constructor term.*

Clearly, all c-terms are constructor terms.

Example

`nil` and `h::t` are c-terms. `h1::zero::t` is a constructor term but not a c-term. □

Definition 11 *A* pattern[7] *is either a constructor term or a tuple of the form* $(P_1, \ldots, P_n)$ *where for* $1 \leq i \leq n$, P_i *is a pattern. A* pattern-set *of type* T *is a set* $\{P_1, \ldots, P_n\}$ *where for* $1 \leq i \leq n$, P_i *is a pattern and each* P_i *has the same type* T, *denoted* $P_i : T$. *A pattern-set is said to be* well-defined *if the* P_i, $1 \leq i \leq n$, *exhaustively cover* T *and there are no overlapping patterns.*

The CHANGE TYPE command can be viewed as a mapping, Θ, from a pattern-set to a pattern-set. Let Ω denote the class of all well-defined pattern-sets. If CHANGE TYPE is used to change a type V_1 to V_2, then the mapping will be denoted as Θ_{V_1,V_2}, although the subscripts will be omitted if they are obvious from the context. This section gives a definition of Θ and then a proof that if $\mathcal{P}$ is a well-defined pattern-set, then so is $\Theta(\mathcal{P})$, i.e. that Θ is defined as:

$$\Theta : \Omega \rightarrow \Omega \tag{7.1}$$

Definition 12 *Suppose a constructor* c *has type* $T_1 * \ldots * T_n \rightarrow T_0$. *Then* c *is a base constructor for* T_0 *if* T_0 *is distinct from each* T_i, $1 \leq i \leq n$. *Otherwise,* c *is a step constructor for* T_0. *Let* $\mathcal{B}(T)$, $\mathcal{S}(T)$ *denote the set of base and step constructors respectively for type* T.

[7] The syntactic category, pattern, in the SML Definition is actually more wide-ranging than this definition. The definition given here is restricted to make the following analysis easier to follow. *CYNTHIA* does not fully support tuples so patterns in *CYNTHIA* are pairs of pairs. Again, this makes no real difference to the analysis.

Example

Define constructors for a tree type as `leaf : 'a -> 'a tree` and `node : 'a * 'a tree * 'a tree -> 'a tree`. Then `leaf` is a base and `node` is a step constructor. □

The definition does not adequately deal with many possible ML types. It does not include mutually recursive types or types without a finite number of constructors (such as integers[8]). It is also unclear how to deal with a constructor with a type something like `T0 list -> T0`. Under the current definition, this will be a base constructor although clearly, functions using this constructor could involve (possibly mutually) recursive definitions. See §7.2.5, p. 166 for further comment.

The definition of the following relation is intended to capture an association between the constructors of the old and new datatype. The definition allows for a heuristic choice to be made but does not explicitly define the heuristics.

Let K_1, K_2 be sets with $n_1 = |K_1|$ $(n_1 > 0)$ and $n_2 = |K_2|$. Define a relation Ψ_{K_1,K_2} over $K_1 \times K_2$ as follows.

- If $n_1 = n_2$ then $< x, \phi(x) > \in \Psi_{K_1,K_2}$ for all $x \in K_1$ where ϕ is a 1-1 mapping from K_1 onto K_2.
- If $n_1 < n_2$, let $K_{2_{|n_1}}$ be some subset of K_2 containing n_1 elements. Let ϕ be a 1-1 mapping from K_1 onto $K_{2_{|n_1}}$. Then $< x, y > \in \Psi_{K_1,K_2}$ if and only if

  $y = \phi(x)$ or

  $< x, y > \in \Psi_{K_1,K_2 \backslash (K_{2_{|n_1}})}$.
- If $n_1 > n_2$, let $K_{1_{|n_2}}$ be some subset of K_1 containing n_2 elements. Let ϕ be a 1-1 mapping from $K_{1_{|n_2}}$ onto K_2, then $< x, \phi(x) > \in \Psi_{K_1,K_2}$ for all $x \in K_{1_{|n_2}}$ and $x \notin dom(\Psi_{K_1,K_2})$ for $x \in K_1 \backslash (K_{1_{|n_2}})$.

[8] Integers could be represented by a finite number of constructors but not in a way that is free. This thesis considers the type of integers to have an infinite number of constructors, each integer being a base constructor

Now, let $\theta_{\mathcal{B},V_1,V_2}$ be a relation defined over the set $\mathcal{B}(V_1) \times \mathcal{B}(V_2)$, such that $\theta_{\mathcal{B},V_1,V_2} = \Psi_{\mathcal{B}(V_1),\mathcal{B}(V_2)}$ and some choice of ϕ, $K_{1_{|n_2}}$ and $K_{2_{|n_1}}$ (may be more than one choice) has been made. Similarly, $\theta_{\mathcal{S},V_1,V_2}$ will be a relation over $\mathcal{S}(V_1) \times \mathcal{S}(V_2)$. Again, V_1, V_2 are omitted if obvious from the context. The precise choice of ϕ, $K_{1_{|n_2}}$ and $K_{2_{|n_1}}$ are governed by the heuristics and restrictions in the previous sections. The choice makes no difference to the analysis presented here. Define $\theta_{V_1,V_2} = \theta_{\mathcal{B},V_1,V_2} \cup \theta_{\mathcal{S},V_1,V_2}$. Note that θ_{V_1,V_2} is a relation where the base (step) constructors of V_1 are related to those of V_2 such that all base (step) constructors of V_2 have a pre-image. In the case that $|\mathcal{S}(V_1)| = 0$ or $|\mathcal{S}(V_1)| = 0$ some target constructors may have no pre-image. Hence, these constructors introduced explicitly, along with the appropriate number of arguments, into the target program.

Example

The above definitions are actually quite simple. Intuitively, if $< c_1, c_2 > \in \theta_{\mathcal{B}}$ then when CHANGE TYPE is applied, each pattern with constructor c_1 is replaced with a constructor c_2. Consider an easy example. Suppose the source (base) constructors are:

```
c1 c2 c3
```

and the target (base) constructors are:

```
d1 d2 d3 d4 d5
```

then the relation is such that all the ds are exhausted. One possible choice of ϕ, $K_{1_{|m}}$ and $K_{2_{|n}}$ gives the following members of $\theta_{\mathcal{B}}$:

```
<c1,d1>, <c2,d2>, <c3,d3>, <c1,d4>, <c2,d5>
```

Note that each target constructor appears exactly once in the relation. □

We now extend $\theta_{\mathcal{B}}$, $\theta_{\mathcal{S}}$ to a relation $\theta_{\mathcal{C}}$ on c-terms in the following way. The relation $\theta_{\mathcal{C}}$ is defined over the set $C_{V_1} \times C_{V_2}$, where C_{V_1}, C_{V_2} are the sets of c-terms of type V_1 and V_2 respectively. First, we need the following definition:

Definition 13 *Let $c(x_1, \ldots, x_r)$ be a c-term (or constructor term) of type T. Then x_i is said to be a recursive argument if x_i has type T and is a non-recursive argument otherwise.*

Consider each pair c, d where $< c, d > \in \theta$ and the arity of c is n and the arity of d is m. The following analysis is for step constructors. The case for base constructors is the same except that recursive arguments need not be considered. Suppose that c has arguments $x_1, \ldots, x_n$ and d has arguments $y_1, \ldots, y_m$. Suppose for clarity's sake and wlog. that the recursive arguments of c are $x_1, \ldots, x_j$ and of d are $y_1, \ldots, y_k$ where $j \leq n$ and $k \leq m$. Associate with c and d a heuristic that makes the following choices:

- If $j \leq k$, choose a subset Y of $\{1, \ldots, k\}$ with j elements and a 1-1 mapping ϕ_Y from $\{1, \ldots, j\}$ onto Y.
- If $j > k$, choose a subset X of $\{1, \ldots, j\}$ with k elements and a 1-1 mapping ϕ_X from $\{1, \ldots, k\}$ onto X.
- If $n - j \leq m - k$, choose a subset Y' of $\{k+1, \ldots, m\}$ with $n - j$ elements and a 1-1 mapping $\phi_{Y'}$ from $\{j+1, \ldots, n\}$ onto Y'.
- If $n - j > m - k$, choose a subset X' of $\{j+1, \ldots, n\}$ with $m - k$ elements and a 1-1 mapping $\phi_{X'}$ from $\{k+1, \ldots, m\}$ onto X'.

Note that these choices are made once for each pair $< c, d > \in \theta$. The relation between the arguments of c and d is described in the following.

The relation, θ_C, is defined[9] such that:

$$< c(x_1, \ldots, x_n), d(y_1, \ldots, y_m) > \in \theta_C \qquad n, m \geq 1$$

if the following conditions hold:

- c has arity n and d has arity m.
- $< c, d > \in \theta_S$.
- The x_is are distinct, $1 \leq i \leq n$, and the y_js are distinct, $1 \leq j \leq m$.
- If $j \leq k$ then $x_i = y_{\phi_Y(i)}$ for $1 \leq i \leq j$ and for $j < i \leq k$, y_i is distinct from any x.

[9] I do the case for θ_S (the case for θ_B is the same but without considering recursive arguments).

- If $j > k$ then $y_i = x_{\phi_X(i)}$ for $1 \leq i \leq k$ and for $k < i \leq j$, x_i is distinct from any y.
- If $n-j \leq m-k$ then $x_{j+i} = y_{\phi_{Y'}(j+i)}$ for $1 \leq i \leq n-j$ and for $n-j < i \leq m-k$, y_{i+k} is distinct from any x.
- If $n-j > m-k$ then $y_{k+i} = x_{\phi_{X'}(k+i)}$ for $1 \leq i \leq m-k$ and for $m-k < i \leq n-j$, x_{j+i} is distinct from any y.

The side-conditions relating to the distinctness of variables mean that θ_C has potentially an infinite number of elements. For this reason, the relation is modulo the renaming of variables.

Example

Consider the following two c-terms, where upper-case (lower-case) arguments are non-recursive (recursive).

```
c(x1,x2,x3,X4)
d(y1,y2,y3,y4,y5,Y6,Y7)
```

The heuristics define a relation between the two c-terms such that (non-)recursive arguments are related to (non-)recursive arguments. In this case, the heuristics choose the simplest relation. We have $j = 3, k = 5, n = 4, m = 7$. The heuristics choose $Y, \phi_Y, Y', \phi_{Y'}$. Suppose that $Y = \{1,2,3\}, Y' = \{6\}, \phi_Y : \{1,2,3\} \rightarrow \{1,2,3\}$ and $\phi_{Y'} : \{4\} \rightarrow \{6\}$ where $\phi_Y(i) = i, 1 \leq i \leq 3$ and $\phi_{Y'}(4) = 6$. Then the following holds (assuming $< \mathtt{c}, \mathtt{d} > \in \theta_S$):

```
<c(x1,x2,x3,X4),d(x1,x2,x3,y4,y5,X4,Y7)> ∈ θC
```

The way to view this is that under CHANGE TYPE, `c(x1,x2,x3,X4)` is replaced by `d(x1,x2,x3,y4,y5,X4,Y7)` where `y4`, `y5` and `Y7` are fresh variables. □

θ_C is defined in such a way that each member of θ_C determines how a particular c-term should be modified under CHANGE TYPE — in fact, it describes how a c-term of the old datatype is related to a set of c-terms of the new datatype. θ_C will be extended later to constructor terms and then pattern-sets.

There will usually be a choice as to how to modify $c(a, b)$ for a given constructor pair c, d. The strategy is governed by the list of heuristics given in §7.2.2. Often, however,

the heuristics will not restrict us to a single choice and so, wlog., an arbitrary decision is made. The heuristics are ignored in what follows.

Recall from §7.2.1 that nested patterns may require multiple mappings to be derived. There is potentially a very large number of these mappings. For each constructor of V_1, each non-recursive argument of the constructor potentially necessitates a new mapping. If a non-recursive argument has been split in the source program, then a new mapping will be needed. This scenario is captured formally in the following.

First, the following definitions are required.

Definition 14 *Let $c(x_1, \ldots, x_n)$ be a constructor term of type T. Fix an ordering on the elements of $\mathcal{B}(T)$. Then $c(x_1, \ldots, x_n)$ is said to be* base-ground *over T if c is the first element of $\mathcal{B}(T)$ and if $x_1, \ldots, x_n$ are also base-ground.*

Definition 15 *Suppose $< c, d > \in \theta_{T_1,T_2}$ for constructors c, d and types T_1, T_2. Suppose also that $< c(x_1, \ldots, x_n), d(y_1, \ldots, y_m) > \in \theta_{C,T_1,T_2}$. Then the relation θ_{c,d,T_1,T_2} is defined in such a way that $< i, j > \in \theta_{c,d,T_1,T_2}$ (for $i, j \geq 1$) if and only if $x_i = y_j$.*

So θ_{c,d,T_1,T_2} describes the permutations of the arguments of c under θ_{C,T_1,T_2}.

Now, extend θ_C to the constructor term case. Define a new relation Θ_{C,T_1,T_2} over constructor terms of T_1 and T_2. This relation recursively reduces the constructor term case to the c-term case.

$$< c(x_1, \ldots, x_n), d(y_1, \ldots, y_m) > \in \Theta_{C,T_1,T_2} \qquad n, m \geq 0 \tag{7.2}$$

if the following hold (assume the same definitions of j and k as with the definition of θ_C):

- c has arity n and d has arity m.
- $c(x_1, \ldots, x_n) : T_1$ and $d(y_1, \ldots, y_m) : T_2$.
- $< c, d > \in \theta_{T_1,T_2}$.
- If $< i, j > \in \theta_{c,d,T_1,T_2}$ for $1 \leq i \leq n, 1 \leq j \leq m$, and x_i is a non-variable, then $< x_i, y_j > \in \Theta_{C,H_1,H_2}$, where $x_i : H_1$ and $y_j : H_2$.

- If $< i, j > \in \theta_{c,d,T_1,T_2}$ for $1 \leq i \leq n, 1 \leq j \leq m$, and x_i is a variable, then $x_i = y_j$.
- If j is such that $\nexists i$ with $< i, j > \in \theta_{c,d,T_1,T_2}$ then y_j is a distinct variable.
- If i is such that $\nexists j$ with $< i, j > \in \theta_{c,d,T_1,T_2}$ and if $x_i : H$ then the top-level constructor of x_i is either base-ground over H or is a variable.

and $< v, v > \in \Theta_{C,T_1,T_2}$ for all variables v and all types T_1, T_2 (note that it is only the variable *names* that are the same, not their types).

The final condition is needed to prevent there being a potentially infinite number of elements in the relation. The condition corresponds to the case where an argument of a constructor is "lost" under CHANGE TYPE. For example, suppose the source pattern-set contains `node(zero,x,y)` and `node(s n,x,y)` then under a change of type that sends `node` to `s` with $< 2, 1 > \in \theta_{\texttt{node,s,nat tree,nat}}$, there does not exist j such that $< 1, j > \in \theta_{\texttt{node,s,nat tree,nat}}$. Hence, both `node(zero,x,y)` and `node(s n,x,y)` are transformed to `s x`. To prevent there being two copies of `s x`, `node(s n,x,y)` is "lost". `node(zero,x,y)` is kept because `zero` is the base-ground constructor of `nat`.

Example

Imagine if the user specified a type change from `nat list` to `('a list) tree`. Consider the claim below.

$$< (\texttt{s n}) :: \texttt{t}, \texttt{node}(\texttt{h} :: \texttt{n}, \texttt{t}, \texttt{t2}) > \in \Theta_{C,\texttt{nat list},(\texttt{'a list})\ \texttt{tree}} \tag{7.3}$$

To see that this is true, note the following. $< ::, \texttt{node} > \in \theta_{\texttt{nat list},(\texttt{'a list})\ \texttt{tree}}$ and the elements of $\theta_{::,\texttt{node,nat list},(\texttt{'a list})\ \texttt{tree}}$ are $< 1, 1 >$ and $< 2, 2 >$. Hence, we need to show that $< \texttt{s n}, \texttt{h} :: \texttt{n} > \in \Theta_{C,\texttt{nat},\texttt{'a list}}$, that `t` = `t` and that `t2` is a distinct variable. Proceeding recursively, the claim is shown to be true. □

Now extend Θ_C to patterns. From definition 11, the general form of a pattern is $(P_1, \ldots, P_n)$ where the P_i is either a constructor term or a pattern. It may be assumed, wlog., that CHANGE TYPE only affects one constructor term P_{ij}, which may be at any depth within $(P_1, \ldots, P_n)$, at a time. Then:

$$< (P_1, \ldots, (\ldots, P_{ij}, \ldots), \ldots, P_n), (Q_1, \ldots, (\ldots, Q_{ij}, \ldots), \ldots, Q_n) > \in \Theta_{C,V_1,V_2}$$

if and only if

$$< P_{ij}, Q_{ij} > \in \Theta_{C,H_1,H_2}$$

where P_{ij} is at any level of nesting, Q_{ij} is at the same level of nesting, $(P_1, \ldots, P_n)$ has type V_1 and P_{ij} has type H_1.

In all that follows, I only do the case when a pattern is a constructor term. The analysis could be extended to the case when a pattern is of the form $(P_1, \ldots, P_n)$.

Finally, Θ_C can be extended to pattern-sets. Define a function Θ_{V_1,V_2}, which takes a pattern-set of type V_1 and returns a pattern-set of type V_2, as follows.

Suppose $P = \{P_1, \ldots, P_n\}$ is a pattern-set of type V_1. Suppose also that for each $1 \leq i \leq n$, there exists a set $\{Q_{i1}, \ldots, Q_{im_i}\}$ (where possibly m_i is zero) such that for all $1 \leq j \leq m_i$, $< P_i, Q_{ij_i} > \in \Theta_{C,V_1,V_2}$. Then, define Θ_{V_1,V_2} such that:

$$\Theta_{V_1,V_2}(\{P_1, \ldots, P_n\}) = \{Q_{11}, \ldots, Q_{1m_1}, \ldots, Q_{n1}, \ldots, Q_{nm_n}\}$$

Example

Refer again to the example at the end of §7.2.2, p. 150. Since `NEG` $\mapsto$ `NEG` and `NEG` $\mapsto$ `LABEL`, we have $<$ `NEG p, NEG p` $> \in \theta_C$ and $<$ `NEG p, LABEL (p, e1)` $> \in \theta_C$.
Hence, $\Theta(\{$`ATOM n, NEG p, AND (p, q)`$\}) = \{$`ATOM n, NEG p, LABEL (p, e1), OR (p, q)`$\}$. □

7.2.4 Correctness of CHANGE TYPE

This concludes the presentation of the CHANGE TYPE algorithm. I now go on to show that the algorithm is correct, in the sense that given a well-defined pattern-set, CHANGE TYPE will modify this pattern-set to produce a new pattern-set which is itself well-defined. Some lemmas will be needed before this result can be shown.

Let $\mathcal{C}$ be the set of well-typed c-terms.

Lemma 1 *If* $< x, y > \in \theta_{C,V_1,V_2}$ *and* $x \in \mathcal{C}$, *then* $y \in \mathcal{C}$.

Proof We first need to show that y is a c-term. By construction, if y is of the form $d(y_1, \ldots, y_m)$ and x is $c(x_1, \ldots, x_n)$, then each y_i is equal to an x_j or is a distinct

variable. Since x is a c-term, each x_j is a distinct variable and hence y is a c-term. Also, by definition, d has arity m and so y is well-typed. ∎

Lemma 2 *For all types V_1, V_2, if $|\mathcal{B}(V_1)| > 0$, $\forall y \in \mathcal{B}(V_2)\ \exists x \in \mathcal{B}(V_1)$ such that $< x, y > \in \theta_{\mathcal{B},V_1,V_2}$. Similarly, for $\mathcal{S}$.*

Proof. By construction. ∎

Lemma 3 *Suppose $|\mathcal{B}(V_1)| > 0$ and $y \in \mathcal{B}(V_2)$. Then there exists at most one $x \in \mathcal{B}(V_1)$ such that $< x, y > \in \theta_{\mathcal{B},V_1,V_2}$. Similarly, for $\theta_{\mathcal{S}}$.*

Proof. Follows immediately from the definition of $\theta_{\mathcal{B}}$ ($\theta_{\mathcal{S}}$). ∎

Lemma 4 *If $|\mathcal{B}(V_1)| > 0$, for each constructor with result type V_2, there is a unique pre-image of θ_{V_1,V_2} that is a constructor with result type V_1.*

Proof Follows immediately from Lemmas 2 and 3. ∎

Definition 16 *Let D be a constructor term of some type V. Then D' is a generalisation of D if $D = D'$ or if D' is obtained from D by replacing any (non-variable) subexpression of D with a variable.*

Example

The generalisations of `node(x,y,node(z,t,leaf n))` are the expression itself, `node(x,y,e1)`, `node(x,y,node(z,t,e1))` and `e1`.

In all that follows, I assume, wlog. that $|\mathcal{B}(V_1)| > 0$ and $|\mathcal{S}(V_1)| > 0$ for all types V_1, V_2 considered. If either of these quantities is zero, then there are some target constructors with no corresponding source constructors. These target constructors are just added explicitly in the target pattern-set. Hence, their inclusion does not comprise the well-definedness of the target pattern-set. The proofs of the following lemmas and theorem could be extended to deal with this degenerate case.

Lemma 5 *For any types V_1, V_2, and any constructor term, D, of type V_2, there exists a generalisation of D, D', and a constructor term, C, of type V_1 such that*

$$< C, D' > \in \Theta_{C,V_1,V_2}$$

Proof The proof is by induction on the depth k of the expression tree of D.

$k = 1$: D must be either a variable or a constructor of zero arity. If D is a variable, then $< D, D > \in \Theta_{C,V_1,V_2}$ so $C = D = D'$. Otherwise, by Lemma 4, there is a unique constructor, c, of type V_1 such that $< c, D > \in \theta_{V_1,V_2}$. Hence, by the definition of Θ_{C,V_1,V_2}, $< c(x_1, \ldots, x_s), D > \in \Theta_{C,V_1,V_2}$ where s is the arity of c.

$k > 1$: Assume that the lemma holds for constructor terms with a depth less than k. Suppose also that D is of the form $d(y_1, \ldots, y_r)$. By Lemma 4, there is a unique constructor, c, of type V_1 such that $< c, d > \in \theta_{V_1,V_2}$. For $1 \leq j \leq r$, y_j is a constructor term of depth less than k. Suppose that y_j has type V_{2j} and c has arity s. Wlog, assume that for $1 \leq j \leq w$, $\exists i$ such that $< i, j > \in \theta_{c,d,V_1,V_2}$ where $1 \leq w \leq r$ and $1 \leq i \leq s$, and that for $w+1 \leq j \leq r$, $\nexists i$ with $< i, j > \in \theta_{c,d,V_1,V_2}$. By the induction hypothesis, for $1 \leq i \leq w$, there exist constructor terms $x_i : V_{1i}$ and generalisations y'_i of y_i such that $< x_i, y'_i > \in \Theta_{C,V_{1i},V_{2i}}$. By the definition of Θ_C, for $w + 1 \leq i \leq s$, x_i must be either a variable or base-ground. For $w+1 \leq j \leq r$, let e_j be a variable. Then by the definition of Θ_C, $< c(x_1, \ldots, x_w, x_{w+1}, \ldots, x_s), d(y'_1, \ldots, y'_w, e_{w+1}, \ldots, e_r) > \in \Theta_{C,V_1,V_2}$. Note that $d(y'_1, \ldots, y'_w, e_{w+1}, \ldots, e_r)$ is a generalisation of $d(y'_1, \ldots, y'_r)$. ∎

Definition 17 *Let $c(x_1, \ldots, x_n)$, $d(y_1, \ldots, y_m)$ (where n, m may be zero) be non-variable constructor terms of type V. Then $c(x_1, \ldots, x_n)$* matches *$d(y_1, \ldots, y_m)$ if $c = d$ and $n = m$ and for $1 \leq i \leq n$, x_i is a variable or x_i matches y_i. A variable z matches any expression. If an expression P matches expression Q, we may say that Q is matched by P.*

Definition 18 *Let P and Q be constructor terms. P and Q overlap if there exists some expression e such that e is matched by P and Q.*

Definition 19 *A pattern-set $\{P_1, \ldots, P_n\}$ over type T is exhaustive if for any expression e of type T, e is matched by one of P_i, $1 \leq i \leq n$.*

Lemma 6 *Let V_1, V_2 be any types. Suppose that $< P_1, Q_1 > \in \Theta_{C,V_1,V_2}$ and $< P_2, Q_2 > \in \Theta_{C,V_1,V_2}$ and that Q_1, Q_2 overlap. Then P_1 and P_2 must overlap.*

Proof (sketch) The proof is by induction on the depth of the expression tree of Q_1.

$k = 1$: Q_1 is a variable or a constructor symbol of zero arity. In the former case, P_1 is also a variable so P_1, P_2 overlap. In the latter case, either Q_2 is a variable, in which case so is P_2 so we are done, or $Q_2 = Q_1$. In this case, by the definition of Θ_C, P_1 is of the form $c(x_1, \ldots, x_s)$ and P_2 is of the form $c(x'_1, \ldots, x'_s)$ where all the xs are variables or base-ground. By instantiating all the variables in $c(x_1, \ldots, x_s)$ to base-ground expressions, we obtain an expression which is matched by both P_1 and P_2. Hence, P_1 and P_2 overlap.

$k > 1$: Similar to the proof of Lemma 5, it can be shown that if Q_1 is of the form $d(y_1, \ldots, y_r)$ then P_1 is of the form $c(x_1, \ldots, x_w, x_{w+1}, \ldots, x_s)$ for some $1 \leq w \leq s$ where $< x_i, y'_i > \in \Theta_{C,V_{1i},V_{2i}}$ for $1 \leq i \leq w$ and x_i is a variable or base-ground for $i > w$. Similarly, if Q_2 is of the form $d'(z_1, \ldots, z_q)$ then P_2 is of the form $c'(x'_1, \ldots, x'_{w'}, x'_{w'+1}, \ldots, x'_{s'})$ for some $1 \leq w' \leq s'$ where $< x'_i, z_i > \in \Theta_{C,V'_{1i},V'_{2i}}$ for $1 \leq i \leq w'$ and x'_i is a variable or base-ground for $i > w'$. Since Q_1, Q_2 overlap, $d = d'$ and hence $c = c'$ by Lemma 4. Hence, $r = q$, $s = s'$ and $w = w'$. For $1 \leq i \leq w$, since y_i and z overlap, by the induction hypothesis, so must x_i and x'_i. Form an expression with c as top-level constructor, the overlap of x_i and x'_i as ith argument for $1 \leq i \leq w$ and instantiate $(i > w)$ variables in x_i with base-ground expressions to make an expression which is matched by both P_1 and P_2. Then P_1, P_2 overlap. ∎

Lemma 7 *Let V_1, V_2 be any types. Suppose that $< P_1, Q_1 > \in \Theta_{C,V_1,V_2}$ and $< P_2, Q_2 > \in \Theta_{C,V_1,V_2}$. Then if P_1 matches P_2, Q_1 matches Q_2.*

Proof The proof is inductive and follows the definition of Θ_{C,V_1,V_2}. The key point to note is that Q_1 may have more arguments than P_1 but Θ_{C,V_1,V_2} says that these must be variables so the matching property still holds. ∎

Theorem 1 *Θ_{V_1,V_2} is a function from Ω to Ω.*

Proof Let $P = \{P_1, \ldots, P_n\} \in \Omega$ be a pattern-set of type V_1. It needs to be shown that $\Theta(P)$ contains no overlapping patterns and exhaustively covers V_2. Suppose that $\Theta(P) = \{Q_1, \ldots, Q_m\}$ for patterns $Q_1, \ldots, Q_m$.

First show that there are no overlapping target patterns. The proof is by contradiction. Suppose that two patterns, say Q_1 and Q_2, overlap. By the definition of Θ_{V_1,V_2}, there exist i, j with $1 \leq i, j \leq n$ such that $< P_i, Q_1 > \in \Theta_{C,V_1,V_2}$ and $< P_j, Q_2 > \in \Theta_{C,V_1,V_2}$. By Lemma 6, P_1 and P_2 overlap. This is a contradiction.

Now show that $\{Q_1, \ldots, Q_m\}$ exhaustively covers V_2. Take any constructor term y of type V_2. By Lemma 5, there exists a generalisation of y, y', and a constructor term x of type V_1 such that $< x, y' > \in \Theta_{C,V_1,V_2}$. Since P is well-defined, x is matched by P_i for some $1 \leq i \leq n$. Now, there is at least one Q_j with $< P_i, Q_j > \in \Theta_{C,V_1,V_2}$. (If there were none, then one of the constructors, c occurring in P_i must be such that $< c, d > \notin \theta_{V_1,V_2}$ for any d. But since the constructors of P_i are a subset of those in x and since $< x, y' > \in \Theta_{C,V_1,V_2}$, this cannot be true). By Lemma 7, since P_i matches x then Q_j matches y'. Since y' is a generalisation of y, Q_j must also match y. Hence, every constructor term of type V_2 is matched by a pattern in $\{Q_1, \ldots, Q_m\}$. ∎

7.2.5 Limitations of CHANGE TYPE

The current state of CHANGE TYPE supports a wide variety of datatypes. For unsplit arguments, CHANGE TYPE can be used for any type. For split arguments, CHANGE TYPE can be used for inductively-defined datatypes that satisfy the restriction in §4.7.1, p. 86. The command still works for many types lying outside this definition but may not produce the optimal result.

This section describes situations where CHANGE TYPE may fail or produce a sub-optimal result. All comments refer to the application of CHANGE TYPE to a split argument. If the argument is unsplit, then CHANGE TYPE will always succeed.

Polymorphic Types

Clearly, pattern matching cannot be applied to an argument with a purely polymorphic type, such as `'a`. It may be that the user specifies a change of type that would require such a phenomenon. In such cases, CHANGE TYPE will fail. As an example, consider the following piece of code:

```
nat list -> int
fun countzeros nil = 0
|   countzeros (zero::t) = 1 + countzeros t
|   countzeros (s n :: t) = countzeros t;
```

If the user changed the indicated type to `'a list`, then it makes no sense to retain two clauses for the non-empty list case because an argument of type `'a` does not have constructors. *CYNTHIA* could detect this and remove, say, the third clause, but this might not be what the user intended. Hence, CHANGE TYPE just fails. To carry out the change of type, the user would have to invoke REMOVE PATTERN first to merge the second and third clauses. After this, the change of type would go through.

This situation occurs whenever the user attempts to change to a type that does not have an ML representation based on a finite number of constructors — such as function types etc. In the case of pair types, *CYNTHIA* could choose one of the components of the pair and base the change of type around its constructors. However, this has not been done. The command also fails for integers because integers are considered as having an infinite number of constructors (one for each integer). Integers can be represented with a finite number of constructors but not in a way that they are free. See §7.3.1 for an in-depth discussion.

What constitutes a recursive argument?

Definition 12 states that a recursive argument of a constructor is one which has the same type as the result type of the constructor. However, there are datatypes that are recursive in nature but that do not satisfy this definition, such as the following datatype for terms:

```
datatype term = Var of string | Atom of string | Term of string * term list;
```

The second argument of `Term` is essentially recursive. Consider the following program:

```
term -> int
fun count (Var str) = 1
|   count (Atom str) = 1
|   count (Term(str,l)) = sum (map count l);
```

This is a problematic definition for *CYNTHIA*. First of all, it is non-trivial to show termination. Recall from Chapter 6 that Walther Recursion uses a size measure based on the recursive arguments of constructors. It uses the same definition of recursive argument as CHANGE TYPE. In this example, however, this definition would not be enough because the second argument of `Term` would not be classed as recursive.

As regards CHANGE TYPE, it may well be that the user tries to produce this definition from a simple count function on lists:

```
'a list -> int
fun count nil = 1
|   count (h::l) = 1 + count l;
```

Consider what happens if the user changes the type from `'a list` to `term`. *CYNTHIA* will try to find a relation between the two datatype definitions, but the constructor `Term` will not count as a step constructor since according to *CYNTHIA*'s definition, it has no recursive arguments. Since *CYNTHIA* insists that base constructors relate to base constructors, `nil` will be related to all three of the constructors in the target datatype. The resulting program will hence be:

```
fun count (Var str) = 1
|   count (Atom str) = 1
|   count (Term(str,l)) = 1;
```

This is a perfectly valid definition and the user could go on to modify this as required. However, it seems as though we have unnecessarily lost potentially useful information from the source program — in particular, a valid recursive call. It makes no sense to copy this recursive call across as it is, since `count l` would be ill-typed if it appeared on the RHS of the third clause in the target program. Instead, *CYNTHIA* should recognise that `l` is a list and suggest a recursive call `map count l`. Since `map` is commonly used to carry out recursion in this way, the user would probably find this new recursive call very useful. To do this, however, would require *CYNTHIA* to revise its definition of recursive argument. Feasibly, *CYNTHIA* could detect that `Term` takes a list of `terms` and be ready to introduce `map` as necessary. This technique could perhaps

be generalised to other types — for instance, if `Term` had a second parameter of type `term tree` — by introducing a map function for the relevant datatype.

Definition 12 also excludes mutually recursive types, such as the following example:

```
datatype mut1 = a | b of mut2
         and
         mut2 = c | d of mut1;
```

7.3 Making and Removing Patterns

The commands MAKE PATTERN and REMOVE PATTERN have much simpler underlying mechanisms than CHANGE TYPE. Neither of these commands make any changes to the specification. They both, however, have entries in `Proof_changes` for changing the relevant IND rule. Recall that an IND rule represents the pattern definition of a function along with the recursion scheme in the definition. The two commands in this section modify the IND rule in some way — generally by adding or removing patterns. In addition, the commands must modify the structure of the proof tree so that branches corresponding to removed pattern clauses are also removed, and branches, corresponding to new pattern clauses, are added.

7.3.1 MAKE PATTERN

The simplest of the two commands is MAKE PATTERN. The user specifies, via highlighting an expression, which variable should be split into patterns. *CYNTHIA* then determines the type of this variable and modifies the IND rule, replacing the variable by a number of cases — one for each constructor of the datatype. In this way, MAKE PATTERN always produces the least complex, well-defined pattern. More complicated patterns can be built up by applying MAKE PATTERN incrementally.

Example

Suppose a synthesis proof contains the following IND rule:

$$\frac{\begin{array}{rcl} \ldots & \vdash & a_{b_1} : ((f\ \texttt{nil}) : T_0 \wedge A(\texttt{nil})) \\ \ldots, X : A(t), \ldots & \vdash & a_{s_1} : ((f\ (h :: \boxed{\texttt{t}})) : T_0 \wedge A(h :: t)\ \wedge \ldots \end{array}}{\ldots L : list \ldots \ \vdash\ ind(L, a_{b_1}, \lambda h.\lambda t.\lambda X.a_{s_1}) : ((f\ L) : T_0 \wedge A(L))}$$

If MAKE PATTERN is applied to the indicated occurrence of `t`, *CYNTHIA* replaces `t` by a case for when `t` is `nil` and a case when it is of the form $h2 :: t2$. In the proof, the IND rule is replaced by a new IND rule:

$$\frac{\begin{array}{rcl} \ldots & \vdash & a_{b_1} : ((f\ \mathtt{nil}) : T_0 \wedge A(\mathtt{nil})) \\ \ldots, X_1 : A(\mathtt{nil}), \ldots & \vdash & a_{s_1} : ((f\ (h :: \mathtt{nil})) : T_0 \wedge A(h :: \mathtt{nil}) \wedge \ldots \\ \ldots, X_2 : A(h2 :: t2), \ldots & \vdash & a_{s_2} : ((f\ (h :: h2 :: t2)) : T_0 \wedge A(h :: h2 :: t2) \wedge \ldots \end{array}}{\begin{array}{rcl} \ldots L : list \ldots & \vdash & ind(L, a_{b_1}, \lambda h.\lambda X_1.a_{s_1}, \lambda h.\lambda h_2.\lambda t_2.\lambda X_2.a_{s_2}) \\ & & : ((f\ L) : T_0 \wedge A(L)) \end{array}}$$

Note how the original IND rule has been modified by a simple substitution. The new IND rule is valid because substitution preserves well-foundedness of induction rules. □

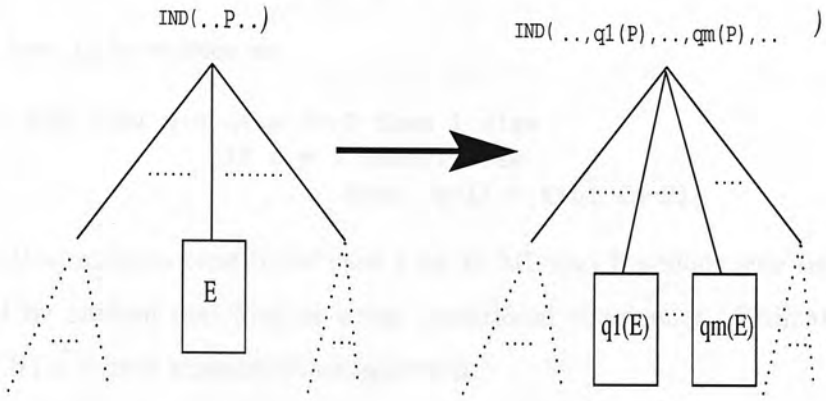
In the example, the target IND rule has one more clause than the source IND rule. This means that an extra branch must be added to the proof tree below the IND rule. The general procedure for doing this is indicated, with an example, in Figure 7.3. If `P` is the argument of the source IND rule being changed, then `P` is replaced by m new cases, `q1(P)`,...,`qm(P)`, where `q1`,...,`qm` are substitutions directly corresponding to the application of MAKE PATTERN. `E` is the branch of the proof tree corresponding to `P`. Note that this has been copied m times, with the substitution `qi` being applied to the ith copy. In the example, the second clause in the source proof is copied to become the second and third clauses in the target.

For each new target constructor term, *CYNTHIA* may add a new recursive call to the list of valid recursive calls. In the example, the user may wish to make use of the recursive call `f t2`. For each new constructor term, *CYNTHIA* adds recursive calls applied to a strict subexpression of the constructor term. Hence, `f t2` is added in this example. The new recursive calls will not affect termination because the strictness means that the measure decreases.

It is obvious that any pattern-set produced by an application of MAKE PATTERN is well-defined because the command adds one case for each constructor of the datatype.

Limitations of MAKE PATTERN

MAKE PATTERN can, of course, only be applied to variables whose type is made up of




```
fun f nil = nil
| f (h::t) = t @ f t;
```

```
fun f nil = nil
| f (h::nil) = nil @ f nil
| f (h::h2::t2) = (h2::t2) @ f (h2::t2);
```

Figure 7.3: The Effect of MAKE PATTERN on the Synthesis Proof.

a finite number of constructors. This includes all user-defined types[10]. Pair types can be included by replacing the variable with a pair (x, y). It makes no sense to apply MAKE PATTERN to function types or wholly polymorphic types.

Integers

The idea of MAKE PATTERN is to automatically construct patterns that are well-defined. For some datatypes, the approach is insufficient. As an example, integers can be regarded as a collection of an infinite number of base constructors and hence MAKE PATTERN, when applied, cannot produce a clause for each constructor. The only way to guarantee well-definedness for integers is to use conditional statements. This means that functions such as:

```
fun fib 0 = 1
|   fib 1 = 1
|   fib n = fib (n-1) + fib (n-2);
```

[10] Although the current version of *CYNTHIA* does not support user-defined types, MAKE PATTERN is implemented in a way such that it would work for any such type.

would have to be written as:

```
fun fib1 n = if n <= 0 then 1 else
             if n = 1 then 1 else
                    fib1 (n-1) + fib1 (n-2);
```

In practice, integers tend to be used a lot in ML and functions over integers may be defined by pattern matching or using conditional statements. Ultimately therefore, *CYNTHIA* should support either approach.

The problem with allowing definitions such as `fib` is that `fib` does not terminate for negative integers. Unfortunately, programmers often write functions like `fib` where the intention is that the function is not evaluated on a negative integer. *CYNTHIA* is meant to disallow ill-defined or non-terminating programs. As such, definitions like `fib` cannot be constructed using *CYNTHIA*'s inference rules. Moreover, even for non-negative integers, termination cannot be checked directly using Walther Recursion since the measure is only defined for datatypes with a finite number of constructors. Given the usefulness of recursive integer definitions, however, it was felt that *CYNTHIA* should support them.

Due to time constraints, the version of *CYNTHIA* used in the evaluations did not incorporate a flexible mechanism for dealing with integers. MAKE PATTERN, for instance, could not be applied to an integer variable. The approach taken was an ad-hoc one. Included in *CYNTHIA* were a couple of built-in examples of functions with integer recursion, specifically:

```
fun t 0 = 0
|   t n = t (n-1);
```

and

```
fun fib 0 = 1
|   fib 1 = 1
|   fib n = fib (n-1) + fib (n-2);
```

The idea was that if the user wished to use integer recursion, he would edit one of the above definitions. Although these definitions do not terminate for negative integers, they will terminate in the positive case. Hence, the termination restriction in *CYNTHIA* was relaxed slightly in the interests of greater flexibility.

To represent `fib` and `t` as proofs, *CYNTHIA* contained internally modified versions where dummy cases were included for negative integers and a conditional statement was used to avoid over-definedness.

This ad-hoc approach allowed students to write recursive definitions over integers as was required by the course notes. In general, however, a better solution is needed. Since the evaluations, such a solution has been devised. It is presented here. One possibility would be to create a translation algorithm that converts integer patterns into conditional statements or into patterns on natural numbers. Such a translation, however, would blur the connections between the internal proof representation and the program. Instead, therefore, new inference rules were created that more directly reflect the program.

There are two cases to consider. *CYNTHIA* should allow the user to define functions using either conditional statements or pattern matching. By the former, I mean functions such as `fib1`. By the latter, I mean functions like `fib`.

To allow `fib1` to be defined and checked for termination, a new inference rule, INTIND, was introduced. This rule is analogous to the IND rule in that it describes the parameters over which recursion may be applied. The rule is the first rule given in Figure 7.4. When applied in a backwards fashion, the rule sets up a single subgoal which is the same as the current goal except that the recursive calls $f(R_i)$ have been specified and conjuncts are set up to prove that they are measure decreasing. No pattern matching is implemented by INTIND. The tree of proof rules corresponding to `fib1` is given in Figure 7.5. Only relevant rules have been included.

INTIND allows the user to define recursion over integers using conditional statements. The definitions must be shown terminating, of course. A notion of measure must be defined for integers. The obvious measure is to use the size of the integer itself. Hence, $\mathtt{n} - 2 \leq_w \mathtt{n}$ and $\mathtt{n} - 1 \leq_w \mathtt{n}$ can be proved by including rules that define the semantics of minus with respect to $\leq_w$.

$$\frac{H, f(R_1) : T_0, \ldots, f(R_n) : T_0 \;\vdash\; a_s : (f(Y) : T_0 \wedge A(Y) \wedge R_1 <_w Y \wedge \ldots \wedge R_n <_w Y)}{H, Y : int \vdash intind(Y,\, a_s(R_1, \ldots, R_n)) : (f(Y) : T_0 \wedge A(Y))}\ \text{INTIND}$$

$$\frac{\begin{array}{l} H, f(R_{11}) : T_0, \ldots, f(R_{1m_1}) : T_0, X_1 : \theta_1 \;\vdash\; a_1 : (f(Y) : T_0 \wedge A(Y) \wedge R_{11} <_w Y \wedge \ldots \wedge R_{1m_1} <_w Y) \\ \qquad\qquad \vdots \\ H, f(R_{n1}) : T_0, \ldots, f(R_{nm_n}) : T_0, X_n : \theta_n \;\vdash\; a_n : (f(Y) : T_0 \wedge A(Y) \wedge R_{n1} <_w Y \wedge \ldots \wedge R_{nm_n} <_w Y) \end{array}}{\begin{array}{l} H, Y : int \vdash intind(Y, \lambda X_1.a_1(R_{11}, \ldots, R_{1m_1})) : (f(Y) : T_0 \wedge A(Y)) \\ \qquad\qquad \vdots \\ H, Y : int \vdash intind(Y, \lambda X_n.a_n(R_{n1}, \ldots, R_{nm_n})) : (f(Y) : T_0 \wedge A(Y)) \end{array}}\ \text{INTIND2}$$

Figure 7.4: Inference Rules for Integer Recursions.

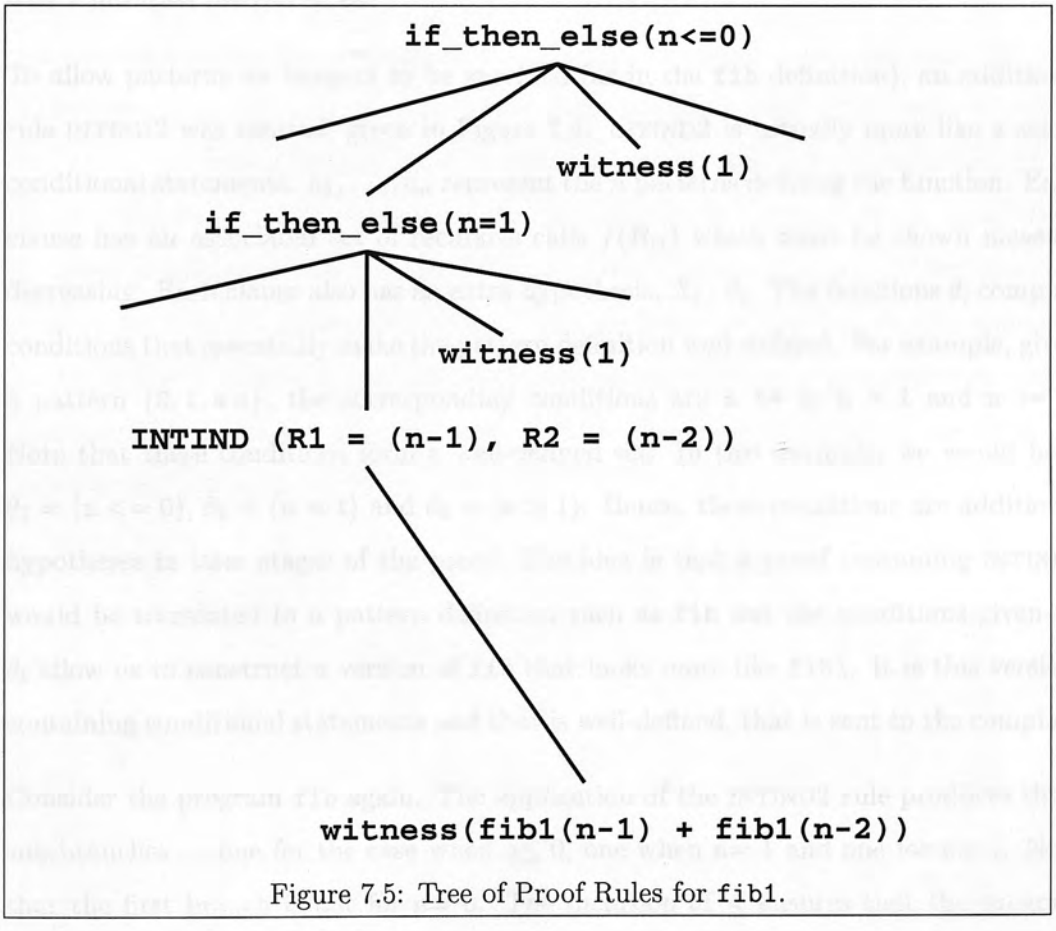


Figure 7.5: Tree of Proof Rules for `fib1`.

Extending the notion of measure is not quite enough, however. Consider the following definition:

```
fun f n = if n <= 0 then f (n-1)
                    else 1 + f (n-1);
```

Although the recursive calls are both measure-decreasing, this function does not terminate. The problem is that the measure for integers has no smallest value. To avoid this kind of thing, a simple side-condition is incorporated into termination analysis. For each recursive argument, X, there must be a corresponding condition of the form $X<=c$ where c is a constant, of type integer. In the first branch of this conditional statement, recursive calls are not allowed. In the second branch, recursive calls must be measure decreasing. This side-condition outlaws the above definition of `f` but would allow `fib1`. Note that it is sufficient for analagous side-conditions to be satisfied if `>=`, `<` or `>` are used instead of `<=`.

To allow patterns on integers to be specified (as in the `fib` definition), an additional rule INTIND2 was created, given in Figure 7.4. INTIND2 is actually more like a set of conditional statements. $a_1, \ldots, a_n$ represent the n patterns defining the function. Each clause has an associated set of recursive calls $f(R_{ij})$ which must be shown measure decreasing. Each clause also has an extra hypothesis, $X_i : \theta_i$. The functions θ_i compute conditions that essentially make the pattern definition well-defined. For example, given a pattern $\{0, 1, \mathtt{s\ n}\}$, the corresponding conditions are `n <= 0`, `n = 1` and `n >= 1`. Note that these conditions form a well-defined set. In this example, we would have $\theta_1 = (\mathtt{n <= 0})$, $\theta_2 = (\mathtt{n = 1})$ and $\theta_3 = (\mathtt{n > 1})$. Hence, these conditions are additional hypotheses in later stages of the proof. The idea is that a proof containing INTIND2 would be translated to a pattern definition such as `fib` but the conditions given by θ_i allow us to construct a version of `fib` that looks more like `fib1`. It is this version, containing conditional statements and that is well-defined, that is sent to the compiler.

Consider the program `fib` again. The application of the INTIND2 rule produces three sub-branches — one for the case when `n`≤ 0, one when `n`$= 1$ and one for `n`> 1. Note that the first branch is not for `n`$= 0$. The inclusion of $\leq$ ensures that the program corresponding to the synthesis proof terminates for negative numbers (assuming the same extensions to Walther Recursion as before).

The inclusion of the INTIND2 rule means that MAKE PATTERN can now be applied to integers. Applying MAKE PATTERN to

```
int -> int
fun fib [ n ]  = n;
```

at the indicated point would give:

```
fun fib 0 = 0
|   fib [ n ]  = n;
```

Similarly, a further application of MAKE PATTERN at the indicated point would give:

```
int -> int
fun fib 0 = 0
|   fib 1 = 1
|   fib n = n;
```

What all of this gives us is a consistent way of representing integer recursion in the synthesis proof. There is a trade-off to be made, however. The proof is no longer semantically equivalent to the program that the user sees — for example, `fib` is non-terminating whereas the program corresponding to the proof does terminate. The idea is that when the *CYNTHIA* code is compiled, it is the internal, well-defined version (i.e. `fib1`) that is sent to the compiler. This means that the likelihood of encountering an error is reduced, but there is a potential cause of confusion because the code that is compiled is different to the code that the user sees. The alternative is to force the user to work with the more specific, internal version, but this also has its disadvantages in that the user is then being restricted to a particular style of programming (and a style foreign to most ML programmers). Further investigation would be needed to see which is the best option.

The extensions described in this section have been implemented. Further extensions are required, however. The integer patterns that can be produced are limited — the user cannot define, for instance:

```
fun f 0 = ...
|   f 2 = ...
|   f n = ... ;
```

Rules have been implemented to deal with the minus function but recursive calls could involve other common functions such as `*`, `/` and `div`. Rules need to be implemented for these functions. Finally, CHANGE TYPE does not yet work for integers in the pattern case.

Catch-all Cases

Another problem with MAKE PATTERN is its inability to introduce "catch-all" patterns. A catch-all pattern is one where a (possibly anonymous) variable is used to deal with more than one case at once. Suppose a datatype has 3 constructors and the user needs to consider combinations of two arguments of this datatype, giving 9 cases in total. Often, a lot of these cases will have the same result and so to avoid code replication specific patterns can be used to specify results for a certain number of cases, and a catch-all pattern is used to define the rest as default. However, this kind of definition cannot be represented in the synthesis proof as all patterns must be non-overlapping. The solution to this problem, which involves expanding the catch-all patterns to give the 9 separate cases, has already been found[11]. Ultimately, it should be incorporated into *CYNTHIA*.

7.3.2 REMOVE PATTERN

REMOVE PATTERN is slightly more intricate than MAKE PATTERN. If the user highlights an expression and selects REMOVE PATTERN, this expression will have been produced in response to a MAKE PATTERN and so will be part of well-defined set of clauses. *CYNTHIA* must be able to identify this set and remove all of its clauses except one. Consider the following set of patterns:

```
nil
zero ::t
(s n)::t
```

If REMOVE PATTERN is invoked at the indicated point, *CYNTHIA* must recognise that the second and third cases form a (well-defined) covering of the natural numbers. *CYNTHIA* would then remove the third clause and replace `zero` by a new variable in the second clause, giving:

```
nil
h::t
```

In this way, *CYNTHIA* removes the covering for natural numbers but leaves a well-defined covering for lists.

[11] personal communication with Konrad Slind

Input: a well-defined pattern-set, P
a clause number, C
an argument number, A
a position, S

Output: a well-defined pattern-set

$newpatts := P$
$key := get\_key(P, C, A, S)$
for each P_i **in** P,
$exp := get\_exp(P_i, key)$
if $exp \neq$ **void**
and $split(exp) =$ **true**
then $newpatts := P \backslash \{P_i\}$
fi
return $update(newpatts, C, A, S)$

Figure 7.6: Algorithm for REMOVE PATTERN.

The algorithm that REMOVE PATTERN uses to decide which clauses should be removed is given in Figure 7.6. C, A and S identify where the user has applied the command. C is the number of the pattern or clause in P. A is the argument number of the function and S is the position of the selected expression with respect to this argument. Given the definition:

```
fun f nil = ...
| f ( zero :: t  1) = ...
| f ((s  zero  2)::t) = ...
| f ((s (s  n  3))::t) = ...
```

box i has clause number $i+1$. A is always 1 because f has only one argument, and S is respectively [], [1, 1] and [1, 1, 1]. The function *get_key* transforms this information into a unique key which represents the same position but with respect to the constructors — so, for example, box 2 has position [1, 1] and key [1/::, 1/s]. This key now contains information about the depth of nesting of patterns. *get_exp* returns the expression within a pattern corresponding to a key. *split* is true if and only if its argument is a non-variable. *update* takes a pattern-set and adds a pattern identical to the first pattern removed except that the constructor selected by the user is replaced by a fresh variable. The basic idea of the algorithm is to identify the well-defined pattern-set to be removed. The key determines this pattern-set. The following example clarifies the situation.

Example

Staying with the current example, consider what happens when REMOVE PATTERN is applied to box (1). The algorithm takes $P = \{\texttt{nil}, \texttt{zero} :: \texttt{t}, (\texttt{s zero}) :: \texttt{t}, (\texttt{s (s n)}) :: \texttt{t}\}$ as one of its inputs. The other inputs determine that the key corresponding to box (1) is []. The algorithm produces a new value for *newpatts* of [] since all four patterns in the pattern-set P have a split expression corresponding to key []. *update* now adds the pattern `x` where `x` is a new variable. The resulting program is:

```
fun f x = ...
```

In the case of box (2), the box has key $[1/::, 1/\texttt{s}]$. Only patterns P_3 and P_4 have split expressions at the position corresponding to this key. Hence, *newpatts* is $\{P_1, P_2\}$. *update* then adds a pattern with a new variable, to give:

```
fun f nil = ...
|   f (zero :: t) = ...
|   f ((s x) :: t) = ...
```

where the third pattern is the one created by *update*. In the case of box (3), there is no change, since n is not split. Note how, depending on which expression the user highlights, the target set of patterns will differ. In all cases, however, the target patterns are guaranteed to be well-defined. □

If any clauses are deleted by REMOVE PATTERN, the corresponding branches in the synthesis proof are also deleted (and hence, the corresponding right-hand sides of equality statements). This situation is analogous to that of adding cases in MAKE PATTERN.

7.4 Summary

This chapter has described the mechanisms underlying the editing commands CHANGE TYPE, MAKE PATTERN and REMOVE PATTERN. These commands can make complex changes to the pattern definition of a function and must be made in a way such that the target patterns are well-defined. A detailed specification was given of CHANGE TYPE, the most complicated of the commands, along with restrictions and heuristics

which govern its use. A proof that it always produces well-defined patterns was also given.

When Mavis had recovered from the shock, she turned round to thank Cynthia. But Cynthia was nowhere to be seen. "That's funny," thought Mavis. "Where can she have gone?" Mavis waited for the best part of an hour and then decided that Cynthia wasn't coming back. "Nothing for it then, I'll take this path." The path was long and curved. Mavis was sure that the path kept going back on itself. She half expected to end up back where she'd started at the end of it all. Still, she trusted Cynthia and so kept going. For an hour. And then two hours. Darkness was beginning to settle in. On she went for another hour. And a fourth. Now it was really black. The forest was starting to make noises. Creaks here and groans there. Mavis began to get frightened. Still, she could do nothing else. And so she followed the path onwards. She'd been at it almost eight hours, when, just as she was about to give up and go back, she turned around a sharp corner. The trees on either side of her cleared away and in the far distance, she finally saw what she had been looking for.

The castle rose up majestically. It had numerous turrets that seemed to reach the heavens. Thanks to the floodlighting, she could see glorious fountains on either side of the castle. Gargoyles reached out from the roofs and intricate designs sprawled over the castle walls. It was quite possibly the most beautiful sight Mavis had ever seen.

Mavis decided to rest. It had been a hectic day. She drifted asleep and dreamt of home.

Chapter 8

Evaluation

This chapter reports on two experimental evaluations of *CYNTHIA* undertaken with a group of students learning ML at Napier University, Edinburgh. The two evaluations took place at different times of the year. The first was a very exploratory evaluation, designed at identifying areas where *CYNTHIA* needed improving and discovering areas where *CYNTHIA* had most impact. In this evaluation, *CYNTHIA* was introduced to the students as a research tool that they could use if they wished but did not have to. The second evaluation was undertaken after some changes, suggested by the first evaluation, had been made to *CYNTHIA*. It was designed to ask some more specific questions about *CYNTHIA*'s use.

8.1 Description of Experiments

8.1.1 The Subjects

In total, there were two evaluations. Both subject groups studied ML as part of a Formal Methods course at Napier University. The course lasted 14 weeks of which approximately 9 weeks was on ML. The students were given lectures each week and were then expected (although not forced) to attend a two hour, supervised tutorial session during which they would work through examples from a Web-based course[1] and could ask questions of the tutors. The ML course is divided into eight short tutorials consisting of the introduction of new concepts and then exercises that the

[1] http://www.dcs.napier.ac.uk/course-notes/sml/manual.html

students could work through. The structure of the course is as given in §2.3 in chapter 2. In previous years, students had used the New Jersey SML compiler (version 0.93) to compile their programs. In the early stages of the course, students tend to write programs directly into the New Jersey interpreter or cut and paste program fragments from the course notes. Later on, they would write programs in a text editor and then compile the program. Assessment on the course was by examination and also practical coursework. The latter involved the students writing a selection of ML functions to achieve some task. The exercise was distributed in week 6 and the hand-in date was week 10.

Evaluation 1 — October / November, 1997

The subjects were 40 postgraduates following a one-year Software Technology course at Napier University. *CYNTHIA* was introduced in the second week of the course. The students were told that *CYNTHIA* was the result of a research project and that they could use it as much or as little as they wished. *CYNTHIA* was only mentioned in passing in lectures. The students were given a questionnaire to assess their programming experience (see Appendices A.1 and A.2). 16 students in subject group 1 (SG1) returned questionnaires. Their programming experience is summarised in Table 8.1, where the numbers represent the students' stated skill level on the scale 1=low, 5=high.

Student	Pascal	SQL	Cobol	C/C++	Oracle	BASIC	LISP	Fortran	Occam	Java	Other
1			3	3							
2						1					1
3											
4	1		1								
5											
6	4		3								
7											
8											
9	3	3									
10	2		3	1	3						
11				4		4					2
12	3	3	4	1		3	2				
13				4				3			
14											
15											
16	2			2							

Table 8.1: SG1 Programming Experience.

The average number of languages experienced is 1.69 and the average skill level is 1.58.

The average number of languages learnt to a level of 3 or above is 1.00.

Evaluation 2 — February – April, 1998

The subjects were 29 students in year 4 or 5 of an undergraduate course in Computer Science at Napier University. *CYNTHIA* was introduced as one of the main teaching tools in the course. The students were not told that *CYNTHIA* was part of a research project. *CYNTHIA* was introduced more fully in the lectures, although details of editing commands and functionality were only taught in the tutorials.

The students' programming experience is given in Table 8.2.

Student	Pascal	SQL	Cobol	C/C++	Oracle	BASIC	LISP	Fortran	Occam	Java	Other
1	4			3						2	4
2	5	3	4	4							3
3	5		5	3		4					
4											
5	3		3	2							
6	3		2	2						3	
7											
8			4	2		3			3	3	3
9					3					2	4
10			3	1							4
11											
12			4								2
13			2	4							2
14			5	5		5					
15			4	4		3					3
16	5		3	5		5				3	
17											
18											
19	5	1	5	3		5					5
20	3	5	3	2						1	2
21			1	4		2					

Table 8.2: SG2 Programming Experience.

The average number of languages is 3.05 and average skill level is 3.93. The average number of languages learnt to a level of 3 or above is 2.19. Clearly, SG2 was more experienced.

8.2 Informal versus Formal Evaluation

It is important to consider which type of evaluation design is most appropriate, both from a logistical point of view and also with respect to the kind of results that should be achieved. One of the main difficulties in evaluating *CYNTHIA* is that the system is

made up of many complementary and interacting features. By evaluating the system in its entirety, it is unclear which features have contributed to the results. On the other hand, the lack of independence of the features mean that they cannot be evaluated in isolation. Traditional ways of evaluating computer systems fall into two camps. The informal approach consists of making observations whilst the system is in use and interpreting these observations according to a pre-defined set of criteria. This kind of design is often used in the formative stages of the system with the intention that recognised deficiences can be fed back into development [Krathwol 93]. The formal approach, on the other hand, attempts to eliminate the interpretative element by making very specific measurements of particular phenomena yielding data which can be analysed statistically. The most common, and generally considered the most reliable, design is the classic control group scenario. Given that the experimenter wishes to investigate some property, P, the subjects are split into two equally sized groups — the experimental group, which is given complete access to the system, and the control group, which is given no access. The changes in P with respect to the two groups can then be measured to see if the system has had any overall effect on P.

The control group design has traditionally been used in agricultural experiments or medical experiments, such as discovering the effect of some new crop treatment or drug. In these situations, the conditions of the experiment can be controlled relatively easily so that internal and external factors cannot overly affect the validity of the experiment. However, when it comes to computer science experiments, where the subject group is a class of students that spend a large amount of time together discussing the teaching methods, it is much more difficult to control the experimental validity. It is for this reason that most of the evaluation of *CYNTHIA* has been at the informal level (although some more formal experiments were also undertaken when they could be adequately controlled). In any case, **Evaluation 1** was intended to be a more exploratory one, the aim being to investigate the kinds of questions that should be asked about the system. **Evaluation 2** was more focussed but the nature of the experimental design meant that formal evaluation was unreliable. The following summarises the main arguments against the formal approach, in the context of evaluating *CYNTHIA*:

- *Ethical considerations.* The students in the subject group in both evaluations were following courses which would directly contribute to their degree mark, partly through the assessment of course work and partly through an end-of-term examination. The use of *CYNTHIA* had the potential to make a significant difference in both the speed of progress and overall understanding of the course (be it a positive or negative difference). It is unethical, therefore, to advantage one half of the students. Even if the effects of *CYNTHIA* were slight, psychological factors could affect the students' performance or attitude towards the course. For example, it would be impossible (as explained below) to keep the two groups separate. As a result, students from the control group could feel like second-class citizens because they are denied access to this new tool.

- *Controlling the experimental setup.* A key factor in the control group approach is to ensure that the two groups are of similar abilities to begin with. Randomization is the only true way to achieve this. However, this is impractical as this would mean that during a particular supervised tutorial session, some students would be using *CYNTHIA* and some would not. This would undoubtedly lead to ill-feeling as described above. **Evaluation 1** provided two groups, each group attending a different tutorial session. Thus, one possibility might be to use one of these sessions as the control, and the other as the experimental group. This is also unsatisfactory, however, as a few students tend to migrate between the tutorials from week to week, depending on other commitments. It is by no means sure, either, that the two groups would be of similar abilities. The course has a number of part-time students who tend to be mature students, with a different background and different abilities to full-timers. Because these part-timers have work commitments, they tend to be grouped into the same tutorials. Even being optimistic and hoping that the two groups would retain similar abilities, this fact would have to be validated by a pre-test. Pre-testing is problematic, though. The students in **Evaluation 1** are postgraduate students from a wide range of backgrounds. Very few have had functional programming experience and some have had no programming experience at all. This makes it very difficult to have a pre-test aimed at assessing programming competence. Even assessing general intelligence is tricky as the students took their degrees at different universities

and so information about their performance is unavailable. Some students are unwilling to divulge this sort of information.

- *Controlling the running of the experiment.* Even given two groups of equal ability, leakage between these groups would be a major complication. Since students are all following the same course, they would communicate with each other between tutorial sessions, and there is no way to stop the control group using *CYNTHIA* outside the tutorials. This would present a major risk to the validity of the results. One alternative would be to stagger the evaluation over two years or more, using the first year's students as control and the second as experimental group. However, this requires that the two year groups are of similar abilities but changes in the admissions procedure in recent years mean that this is not true at Napier University. [Ormerod & Ball 96] describes an evaluation of a Prolog environment comparing groups from three consecutive years, but the authors were able to rely on a uniformity of the groups.

- *Interdependency of CYNTHIA's features.* Formal experiments are often used to isolate a specific aspect of a system and evaluate change of this feature in very controlled circumstances. This tradition goes back to methodologies such as GOMS [Card *et al.* 83] which suggest the close inspection and timing of very detailed tasks such as deleting a word in a text editor. This kind of approach is not suited for evaluating *CYNTHIA*, however. As [Green & Petre 96] says:

> *But that tradition [GOMS] is not suitable for evaluating programming environments. If we tried to evaluate a programming environment that way we would be overwhelmed by a mass of detailed time predictions of every simple task that could be performed in that environment. Even if we had the timings, and could digest them, they would only address a few of the questions that designers ask.*

For these reasons, the majority of experiments undertaken were at an informal level. Further research could concentrate on more formal experiments. The next section describes the methods for collecting data.

8.3 Data Collection Methods

The following data collection methods were employed. SG1 / SG2 indicates which evaluation used the method.

- **Automatic Logging (SG1/SG2).** All interactions with *CYNTHIA* were automatically recorded. The format was that each editing command, along with any parameters entered in dialog boxes, as well as error messages, and the resulting ML function definition were sent to a dribble file. This provides the right level of detail for easy analysis. In fact, students' sessions can easily be played back in *CYNTHIA* at a later stage. The main problem in interpreting the logs is that it is easy to see what has been done, but it can be difficult to interpret why it was done. This is a particular problem when looking to see if the student has understood and acted correctly to an error message. SML-NJ interactions were not usually scripted, except in the crossover experiment (see below).

- **Observation (SG1/SG2).** I acted as a tutor during the supervised tutorials. This gave me a first hand chance to observe students using *CYNTHIA* and, in particular, what problems they were having. SG1 were aware that I was involved in the development of *CYNTHIA*, whereas to SG2, I was just another tutor.

- **Audio-Visual Protocols (SG1).** A small set of volunteers were video-taped whilst writing ML programs. Four volunteers were invited to write some simple ML functions. The students worked through some examples using *CYNTHIA* and some examples using SML-NJ. Each student was given a maximum of 45 minutes. They were videoed while working and were asked to verbalise what they did at each stage. Students were instructed that they could ask questions if they got really stuck. It was intended that students would only be given technical help – for instance, if they could not find the correct *CYNTHIA* editing command, or if they had problems cutting and pasting within SML-NJ. However, students also had to be given some help on ML. This was kept to a minimum although one student had to be given a lot of help. The tasks the students were asked to attempt are given in Appendix A.4.

- **Questionnaires (SG1/SG2).** The questionnaire given to SG1 is in Appendix A.1 and that given to SG2 is in Appendix A.2. Questionnaires are not generally suitable for formal analysis but are ideal for getting feedback about students' attitudes towards *CYNTHIA*. For SG1, 16 out of 40 responses were collected. For SG2, it was 21 out of 29. Both questionnaires were given towards the end of the course.

- **Crossover Experiment (SG1).** This was an attempt at a more formal evaluation. Although the experiment was not altogether successful, some interesting results were achieved and the report here shows how difficult experiments of this kind can be to perform. Volunteers were asked to attend a one and a half hour session. 21 students responded, 17 of which took part in the experiment. Students were split into 2 groups, group A and group B. Both groups were given a (pencil and paper) multiple choice test lasting ten minutes designed to assess their general ability in ML. Group A were then given test X and group B were given test Y. These tests lasted half an hour. Finally, group A were given test Y and group B were given test X. Again, the students had half an hour. Tests X and Y each consisted of three simple list recursive functions that the students were asked to define. The difficulty of test X and Y was similar. Group A attempted test X using *CYNTHIA* and then test Y using just the compiler. Group B attempted test Y using the compiler alone and then test X using *CYNTHIA*. Tests X and Y are given in Appendix A.5.

  Crossover tests such as this are traditionally used where there are ethical barriers to using the simpler control / experimental group tests. In this case, the advantage of a crossover test is that each student tries examples *with* and *without* *CYNTHIA* so that even if no quantitative data was forthcoming, it is possible to compare individual students' performance under the two different environments.

  Unfortunately, tests X and Y turned out to be too difficult for the students given the time available. Out of the total of 6 questions, the mean number of correct answers in group A and B was 0.78 and 0.63 respectively. Most students got 0 or 1 correct solutions. Two students managed to correctly answer 3 questions. Clearly then, it is meaningless to read anything into the comparative success rates

of the students. However, the experiment produced logs of individual students attempting very similar problems using just SML-NJ and using *CYNTHIA*. This turned out to be an invaluable source of comparative information. By counting the number of different kinds of errors, a useful comparison can also be achieved.

- **Focus group (SG2).** Four students took part in a 45 minute "focus group" session where they were asked questions about their experiences with *CYNTHIA*. This provided some good general feedback.

- **Feedback from the Course Lecturer (SG1/SG2).** The course lecturer wrote a small report on the use of *CYNTHIA* in both evaluations. This is reproduced, without editing, in Appendix A.6. I also gleaned his opinion from a number of fruitful discussions.

8.4 Research Questions

This section sets out the main research questions I am trying to answer. The questions have been organised hierarchically. Table 8.3 sets out the the criteria used to assess each question and the measure used to evaluate these criteria. The criteria essentially give explanations of each research question by breaking the question down into sub-questions. Green's cognitive dimensions in the criteria of question 2.a. are a set of criteria for discussing interactive systems — see the section of 2.a. for explanation.

The questions themselves are given below.

1. How successful was *CYNTHIA* as an editor for ML that guarantees correctness?
   (a) How does the quantity of errors made using *CYNTHIA* compare to text editors?
   (b) How does *CYNTHIA*'s error feedback compare to the compiler's?
   (c) How does the user's productivity rate using *CYNTHIA* compare to text editors?
2. How does programming by analogy compare to writing a program from scratch?
   (a) Are the editing commands well-designed?

(b) How easy is it to choose a starting (source) example?

3. How successful was *CYNTHIA* as a practical programming tool?

   (a) What was the attitude of the users?

   (b) Was the interface easy to use?

4. How useful was *CYNTHIA* as an addition to current teaching methods?

   (a) What concepts were learnt that otherwise might not have been and were they picked up easily?

   (b) Was programming style affected?

   (c) Which students benefitted most from the use of *CYNTHIA*?

   (d) How easily was *CYNTHIA* incorporated into existing teaching methods?

Question	Criteria	Measures
1a)	What kind of errors can be made? How many errors are made?	Error counts from logs Observation
1b)	Are errors located easily? Are they corrected quickly?	Error counts Observation Lecturer report Videoing
1c)	Did students get through more examples?	Videoing Observation
2a)	Green's cognitive dimensions	Observation Error counts
2b)	How were examples chosen?	Observation Lecturer report Videoing
3a)	Did they like *CYNTHIA*'s features? Would they use it again?	Questionnaires Focus group
3b)	Is structure editing useful? Was *CYNTHIA* practical enough?	Questionnaires Lecturer report Focus group
4a)	Was termination / well-definedness important?	Videoing Lecturer report Observation Focus group
4b)	Did students follow a particular style (e.g. top down)	
4c)	Was *CYNTHIA* used more by weaker or stronger students?	
4d)	What had to be done to *CYNTHIA*?	

Table 8.3: Research Questions — Criteria and Measures.

8.5 Interacting with *CYNTHIA*

Before I go on to answer the research questions, it is necessary to acquaint the reader with the exact way of interacting with *CYNTHIA* as experienced by the students. Various parts of this interaction will be referred to later. Take a typical example of applying an editing command. Suppose the user is editing the function given in Figure 8.1. The definition contains a type error, highlighted in pink. To remove the type error, the user clicks with the left mouse button on the expression and then selects CHANGE TERM from a menu. A dialog box pops up in which the user may type text as normal to change the expression. This is displayed in Figure 8.2. The user now needs to compile the final version. This is done by selecting 'Save' from the 'FILE' menu which writes a file called `delete.sml`. This can then be loaded into a compiler in the normal way. In SML-NJ, this is done by typing `use ''delete.sml''`. Comments on this procedure are given in answer to question 1.c.

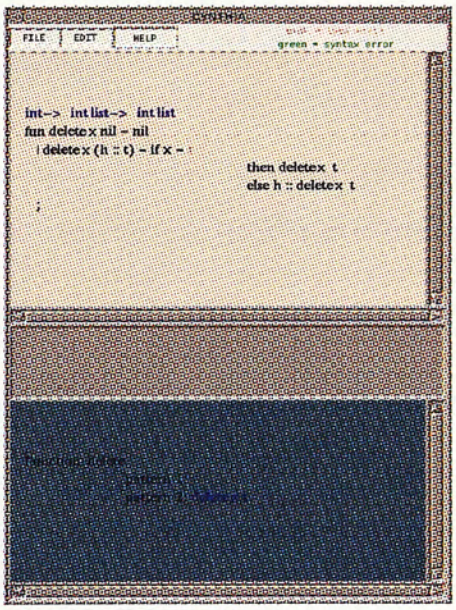


Figure 8.1: Editing `delete` (1).

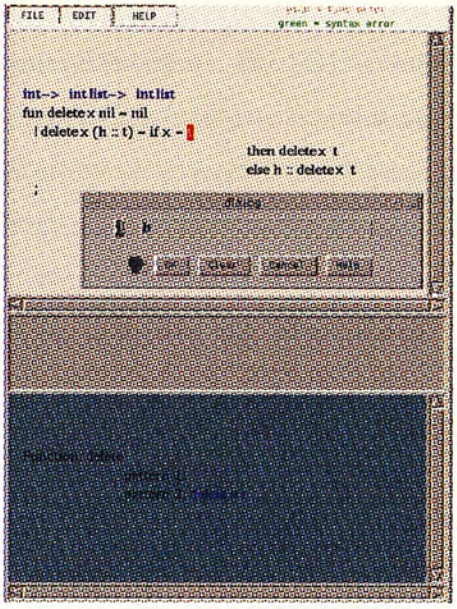


Figure 8.2: Editing `delete` (2).

8.6 Evidence

1. How successful was *CYNTHIA* as an editor for ML that guarantees correctness?

CYNTHIA was primarily designed as an editor that includes sophisticated correctness-checking techniques. This question asks whether it is useful to have a such an environment.

1.a. How does the quantity of errors made using *CYNTHIA* compare to text editors?

Throughout this chapter, *CYNTHIA* will be compared to a traditional text editor. The text editor approach will usually be referred to as TEA, but may also be called the compiler-only approach. Recall that two groups of students were evaluated and SG2 was given a slightly more developed version of *CYNTHIA*. Recall also that the programming experience of the two groups differed significantly and that for SG2, *CYNTHIA* was more integrated into the course.

One way to compare the number of errors made is to look at the logs of *CYNTHIA*-

interaction and make a count of the errors. For SG1, a count was made of errors made during the crossover experiment. The crossover experiment took place over one hour of intense programming. SG2 errors were counted over 8 weeks during which *CYNTHIA* was used regularly by a large number of students for two hours each week (although less intensely). A classification of all errors made was developed, inspiration being drawn from [Aitken 96]. Figure 8.3 gives this classification.

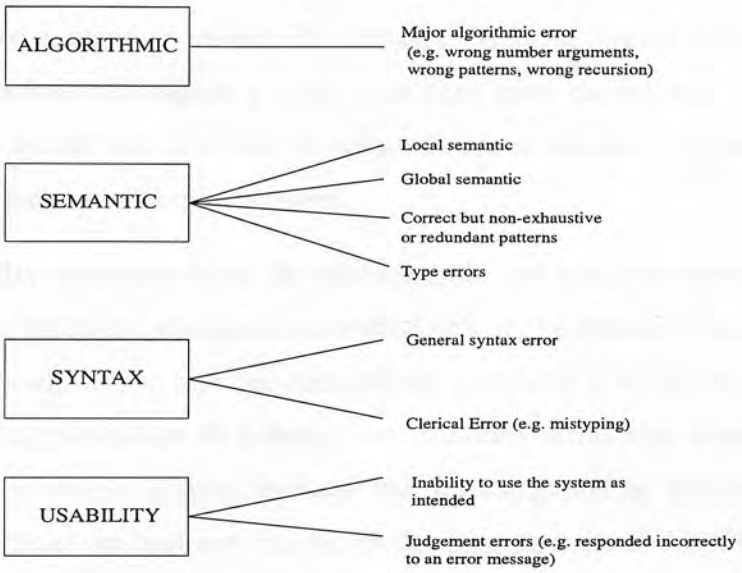


Figure 8.3: Classification of Programming Errors.

I briefly explain the motivation behind this classification. A full list of each error in each class is given in Appendix A.3.

- **Algorithmic** errors suggest a major algorithmic flaw in the program, such as giving the wrong condition in a conditional statement. These errors arise when the user has misunderstood the problem or is unable to design a solution. *CYNTHIA* was not primarily designed to help with this kind of error (although analogy may provide some help), so I do not include a count of these.

- **Semantic** errors are split into four categories. Local semantic errors arise in response to a misunderstanding of part of ML's semantics such as trying to define the type `int string` or overloading a variable. Global semantic errors are where the error is dependent on some other part of the definition — for example, the use of an unbound variable or undefined function. Although type

errors could be seen as global semantic errors, they are given a separate category for emphasis. The same is true of pattern errors (i.e. patterns are overlapping or non-exhaustive). Although a program with pattern errors will successfully compile, they are included because they can be a source of run-time errors and because *CYNTHIA* was designed to forbid them.

- **Syntax** errors include clerical errors where, for example, the student clearly mistyped a name or missed off a bracket. General Syntax errors are slightly more serious and suggest a cause more than mere carelessness : examples are using a syntax that ML does not support such as `return 0`, or using the wrong syntax for a conditional statement.

- **Usability** errors are when the student could not work the system properly. I make no comment whether it is the student's or the system's fault. An example is not being able to find the right editing command in *CYNTHIA* or entering the wrong parameters in a dialog box. Usability errors also include judgement errors: where the system feedback was misunderstood by the student causing him to make an incorrect change to the program. In *CYNTHIA*, this could happen if, for instance, a syntax error is given in a dialog box and the student changes the wrong part of the entry. In TEA, the user might change a clause in the definition which was perfectly correct because he did not realise which clause the error appeared in.

A brief note is needed on how the errors were counted. The errors were counted manually from the logs obtained during the experiment. As explained in chapter 2, automatic counting based on error messages can be inaccurate. For example, in SML-NJ, if the student inputs `1::2 nil`, the system will give a type error because it interprets the input as `1::(2 nil)`. However, it is likely that the student just forgot a '`::`' and meant to enter `1::2::nil`. So the error is really a clerical error. Counting manually also has its problems. First, it has to be decided at which point in the logs errors will be counted. In SML-NJ, each time the student typed ; to evaluate a program attempt, any errors present were noted. In *CYNTHIA*, the program was checked after each editing command was applied, and any errors in the program were counted. In some cases, multiple commands may be required to get the program into

a "consistent" state. For instance, if the user wishes to change a base case output to `nil` rather than `0`, he must do two things: make the actual change, which results in a type error, and then change the result type in the declaration which will eliminate the type error. "Intermediate" errors like this were not counted. Some commands require text to be entered into a dialog box. Any errors present in such textual input were also counted. Second, it can sometimes be difficult to interpret the true reason for an error. For instance, *CYNTHIA* contains as primitive a polymorphic tree type, `'a tree`. Suppose the user enters `tree`, though. Has the student merely omitted the `'a` in a fit of over-zealous typing or does he in fact not realise that the semantics of `tree` require that it takes a parameter? In most cases, the context gives a very large clue to the real cause, but occasionally it is almost impossible to decide.

Table 8.4 gives the error count for the SG1 crossover experiment for both *CYNTHIA* and non-*CYNTHIA* users. Table 8.5 gives the number of errors of SG2 *CYNTHIA* users. It also gives the errors as a percentage relative to the first experiment and the number of errors per edit, expressed as a percentage.

	CYNTHIA	non-*CYNTHIA*
Local Semantic	20	33
Global Semantic	14	21
Patterns	0	6
Type Errors	20	36
General Syntax	0	8
Clerical	9	40
Incorrect Usage	53	0
Judgement	50	53
Total	166	197

Table 8.4: SG1 Errors.

We shouldn't try to draw too many conclusions from the pure quantitative nature of Table 8.4. However, it does seem, both on a quantitative and qualitative level, that the kinds of errors committed by *CYNTHIA* users are *different* from those committed by SML-NJ users.

	No. errors	% relative to expt 1	% Error rate
Local Semantic	39	195	1.19
Global Semantic	41	293	1.26
Patterns	0	0	0
Type Errors	59	295	1.8
General Syntax	0	0	0
Clerical	32	356	0.98
Incorrect Usage	74	140	2.3
Judgement	27	54	0.8
Total	323	175	
Total no. Edits	3266	690	

Table 8.5: SG2 Errors.

Syntax Errors

Syntax errors were almost eliminated when using *CYNTHIA*. *CYNTHIA* is expected to guarantee syntactic correctness. However, syntax errors can still occur in *CYNTHIA* because users may type incorrect syntax in dialog boxes. These errors will never make it through to the final program (although they are still counted). The number of syntax errors has been reduced considerably. In particular, the number of clerical errors has reduced by 78% for SG1.

Semantic Errors

The number of semantic errors also was less for *CYNTHIA*-users. Note particularly that the number of type errors made using *CYNTHIA* was just over half that for non-*CYNTHIA* users. This is as expected. First, fewer type errors should be made because the user is transforming an already well-typed program. Second, once an error does occur, it should be corrected more easily. In some cases, users will recognise there is a type error, but change the wrong part of the program and introduce new type errors. This happens much less in *CYNTHIA*. The raw figures do not tell us how easy it was to correct type errors under each system. However, anecdotal evidence suggests it was much easier using *CYNTHIA* – see question 1.b.

Both the number of local and global semantic errors are fewer with *CYNTHIA* than without. Global errors can manifest themselves in two ways in *CYNTHIA*. First, the user may make reference to, say an unbound variable, in a dialog box. In this

case, *CYNTHIA* rejects the input. The point again, though, is that the error is far more localised than if a text file had been compiled in its entirety. Second, the user may introduce a global semantic error by applying an editing command, such as REMOVE ARGUMENT, where the argument removed is referenced elsewhere in the program. In this case, the error will be highlighted in green (or a single line in the monochrome version) and so, as with type errors, can be located immediately. It is also understandable that local semantic errors are fewer since global semantic errors involve the interaction of possibly far apart program fragments.

Usability Errors

For SG1, *CYNTHIA* seems to score pretty badly on usability. In particular, 53 Incorrect Usage errors were introduced that obviously did not occur when using TEA. This is a disappointingly large figure. It suggests first that *CYNTHIA*'s interface is difficult to use and second that students do not read documentation – for most of the errors they committed could have been avoided if they had read the documentation.

Surprisingly, there is no real difference in the numbers of judgement errors. Anecdotal evidence suggests that students using TEA spend much more time trying to locate errors than *CYNTHIA* users. The judgement errors were meant to measure this sort of thing, but the results do not back up the informal observations. It is worth some closer inspection to see why this is the case. Table 8.6 gives a more fine-grained analysis of the judgement errors. See Appendix A.3 for definitions of J1-J5.

	CYNTHIA	non-*CYNTHIA*
J1	8	0
J2	0	53
J3	35	0
J4	4	0
J5	3	0
Total	50	53

Table 8.6: Judgement Errors made during Crossover Test.

All of the SML-NJ judgement errors fall into the same category – J2. J2 occurs when the student has misunderstood an error message and has changed the wrong part of the program in response to this message. The fact that all SML-NJ judgement errors are

of type J2 shows that the SML-NJ error messages are cryptic. These are fairly serious errors. As expected, these kinds of errors do not occur with *CYNTHIA* because the nature of the editor tells you much more clearly where the error is. However, there are a large number of J3 errors for *CYNTHIA* users. J3 errors are similar to J2 errors but are concerned with dialog boxes. Users may be asked to type in text as a parameter to an editing command. If there is an error, they will be told and should edit the text. A J3 error is when the student edited the wrong part of this text in response to *CYNTHIA*'s error message. Close analysis of the logs show that about 95% of the time, J3 errors occurred because the user tried to type in an entire conditional expression. They might type in `if h=x then delete x t else h::delete x t` and get a syntax error because the `if` keyword is not recognised by *CYNTHIA* (it expected the user to simply type `h=x`). The student would then try and make changes such as changing to `h>x` or `h::(delete x t)`. In fact, one student made 13 such mistakes. Clearly, the real problem here is that the student has not correctly understood the functionality of the command needed. I claim that if the usability of *CYNTHIA* was improved, then these sorts of errors would also disappear. Hence, *CYNTHIA* would provide a much improved method for locating errors quickly and accurately.

To test out this claim, a number of changes were made to improve usability for the second evaluation. This included minor changes to the interface but the attitude was taken that it was not mainly the interface that was at fault, but the documentation of the system. It was assumed, wrongly, in the first evaluation that the editing commands were simple enough for their functionality to be obvious. Hence, the only documentation was limited on-line help, a short tutorial and a Web page describing each command in detail. However, this Web page was at a different site than the Napier course Web page and so was referred to only rarely. For the second evaluation, the documentation of *CYNTHIA* was closely integrated into the Napier course notes — each time a new concept was introduced, the corresponding editing command was introduced also and the student was taken through a couple of examples which specifically used *CYNTHIA*. The intention was that if the students could relate more easily to the set of commands, they would commit fewer errors.

The main change to the actual interface was to give keywords in the dialog boxes that

provided strong hints as to what input was required. Hence, when adding a conditional statement, as in the example above, the dialog box would prompt the user with

```
if [ ] then   else
```

rather than just a box. This made it more obvious what was required.

Note that for SG2 there was no control group, as *CYNTHIA* was introduced as a teaching tool in its own right, not a mere research project. Also, the counts for SG1 were made during a timed session — i.e. over a much shorter period of time. To enable a tentative comparison with SG1, the second column of Table 8.5 gives the increase from experiment 1 to experiment 2 expressed as a percentage of experiment 2 over experiment 1. It gives a very cautious estimate of the change in relative importance of each kind of error. All things being equal, we would expect each error class to increase by the same amount.

Interpolating the figures in Table 8.4, one would expect a much larger number of Incorrect Usage errors than were actually found (in Table 8.5). This is evidence that the improved documentation in the second evaluation meant that students were less likely to commit these kinds of errors.

Judgement errors are the only error that have reduced in absolute terms. This is mainly due to the reduction of J3 errors. In the first experiment, J3 accounted for 70% of the total Judgement Errors. In the second experiment, it is only 16%.

It would be useful to distinguish between different kinds of error messages in *CYNTHIA*. There are basically two kinds of error: type A errors where there really is an error, e.g. in syntax, typing. And type B errors where *CYNTHIA* gives an error because the user is trying to do something that is not possible — e.g. apply MAKE PATTERN to a polymorphic variable. It seemed that the students were often unable to make the distinction between these two things. Hence, they would apply MAKE PATTERN, get an error message, then do some other things before trying MAKE PATTERN again in the hope that it would now work. In these situations, the user has missed the key point — that purely polymorphic variables do not have constructors.

The kinds of errors encountered seemed to change between the two evaluations. In the first evaluation, the majority of errors were type A errors. In the second evaluation,

there were relatively fewer type A errors but more type B errors. This suggests that users were held back less in the early days and managed to progress to deal with more interesting concepts.

In fact, the error classification given in Table 8.4 could be directly fed back into the development of *CYNTHIA*. One of the problems with *CYNTHIA* currently is that the error messages are too generic. Syntax error will be produced for a number of different reasons. I do not think that the SML-NJ compiler fares particularly better, because it just lists all problems it can see without making a judgement about the root cause. It seems to me that Appendix A.3 lists potential root cause problems and that the user's input in *CYNTHIA* could be analysed with respect to these contenders. This is likely to be more successful in *CYNTHIA* because only small chunks of the user's program are parsed / analysed at any one time. Hence, the number of errors at any one time will be fewer and will be localised to the currently highlighted part of the definition.

1.b. How does *CYNTHIA*'s error feedback compare to the compiler's?

The previous question showed that the quantity of errors made using *CYNTHIA* is generally less than without *CYNTHIA*. This question concerns not the amount of errors made, but given that an error has occurred, how easily could the students identify it and correct it? The question can be partly answered with the aid of quantitative data — see Table 8.6 which shows that there are a large number of J2 errors associated with a compiler–only approach. This is because ML compilers tend to produce spurious output once an error has occurred which can be extremely difficult to decipher. **Evaluation 1** also had a large number of judgement errors (J3) associated with *CYNTHIA* but these seem to have been reduced in **Evaluation 2**.

The question can also be answered on a more qualitative level. Evidence from the videoing, observation and communication with students seems to suggest very positively that *CYNTHIA* improves the situation. First, students make fewer errors to begin with, because considerable amounts of code are produced automatically. Hence when an error is encountered the students have more time to think about why it has occurred. Clerical errors, for instance, are greatly reduced by the transformational

approach which vastly reduces the frustration associated with current ML compilers. If an error occurs in *CYNTHIA*, the chances are that it is non-trivial and therefore, worth looking at. In stark contrast to users of the SML-NJ compiler, I noticed students paying attention to error feedback and trying to work through the problem. They did not always succeed, of course, but undoubtedly learnt something along the way. Another point is that it is much easier in *CYNTHIA* to distinguish what *kind* of error is occurring. This is because the error feedback is different for different categories of errors. Chapter 2 described how syntax errors in ML can often have knock-on effects meaning that they show up as type errors during compilation. This does not happen in *CYNTHIA*. The divide between errors is very clear-cut. Type errors are always shown by pink highlighting. Global semantic errors are shown by green highlighting, and syntax errors can only occur in dialog boxes.

Second, students are editing smaller chunks of program at a single time and hence the range over which the error could have occurred is far less. With TEA, students write an entire function before attempting compilation. With *CYNTHIA*, however, as each sub-expression is entered into a dialog box, the text is checked for errors immediately. This means that the student need only look over very small chunks at a time and need worry less about dependencies with other code fragments. In addition, the user knows that some parts of the program are guaranteed correct — for example, any patterns will have been built up using MAKE PATTERN and therefore the patterns must be well-defined. Since some code is generated automatically, there is no reason for the user to suspect an error there. Hence, *CYNTHIA* allows the user to narrow his field of vision when looking for errors.

It seems that the highlighting of ill-typed expressions definitely makes it easier for students to recover from type errors. There were significant differences between SG1 and SG2 in this context. SG1 and SG2 were given slightly different versions of *CYNTHIA* to use. For SG1, given a definition such as:

```
int list -> int
fun maxlist nil = 0
|   maxlist (h::t) = if [ h > [t]2 ]1 then h
                     else maxlist t;
```

[h>t]1 would be highlighted because h and t have different types. For SG2, the feedback

is more closely related to the type-checking tactics. Since `h` has type `int`, it is assumed that `>` has type `int * int -> int` and so it is only `t` that is inconsistent. Hence, only $\boxed{\texttt{t}}^2$ would be highlighted. This is because the type-checking algorithm works left to right so first it would deduce that `h` is an integer and then would expect `t` to be an integer also. It seems that in most cases, the second kind of highlighting is more useful to the student since it narrows down the error location. Of course, the feedback, in this case, is dependent on a left to right bias which could be confusing. However, in most cases type inference is not required because of the existence of the top-level type declaration. This declaration better reflects the user's intentions and so it makes sense that type error highlighting is based on it.

When dealing with type errors, students in SG1 immediately noticed that there is a problem and knew which part of the program to change. They did not always know the correct changes to make, however. Student 2, in the videoing session, trying `maxlist`, had as part of his program:

```
if h > [h::t] then ....
```

Although `h > [h::t]` is highlighted in pink by *CYNTHIA*, the student did not try to change this immediately. He noticed something was wrong but was not capable of making a correction just yet. He went off and edited other parts of the program before finally coming back to it. This seems to be a relatively common phenomenon. Students might ignore the highlighting but would have to address it sooner or later. When student 2 came back to address the type error, he verbalised: "Why is this this colour?" He tried to change the condition to `h > (h::t)` but it remained pink. "`h::t` is a list and what I want is just a number." So, student 2 correctly identified the cause of the type error but was not immediately aware of the correct expression (i.e. which integer) with which to replace `h::t`. After a little more thought, he found the correct answer.

This observation is typical of SG1. Although students always knew where an error had occurred, they often needed help to be able to correct it. In contrast, SG2 were more capable of correcting the error. It is unclear whether the changes in the highlighting mechanism helped, however. SG2 were more experienced in the use of types, having more experience with languages that have the same notion of types, such as C. Even so,

the fact that there was very little trial-and-error when changing highlighted expressions is in stark contrast to what typically happens when students are encountered with SML-NJ type errors. With SML-NJ, students regularly do not know which part of the program is wrong. They make a half-hearted guess and then re-compile the program, only to find more error messages. This process can go on for some time. In *CYNTHIA*, students can just change the pink expression and see what happens rather than having to re-compile everything. It seems then that *CYNTHIA* succeeded in giving the students a more focussed way of correcting their type errors.

There are situations where *CYNTHIA* can be misleading with regards to type errors. This happens when type inference would succeed on a definition but type checking in *CYNTHIA* fails because of extra restrictions placed (unwittingly) on the definition by the type declaration. An example is where the compiler would automatically unify two polymorphic variables but *CYNTHIA* does not. Consider the example:

```
'a -> 'b -> 'a
fun f x y = if g x then x else [ y ] ;
```

The user gets a type error in *CYNTHIA* at the indicated point until he changes `'b` to `'a`. It may be possible for *CYNTHIA* to automatically update such type declarations (or at least suggest updates). In any case, it would be worth working at incorporating type inference in *CYNTHIA*.

1.c. How does the user's productivity rate using *CYNTHIA* compare to text editors?

It is to be expected that users would progress more quickly using *CYNTHIA* for two reasons. First, the editing commands automate certain common programming steps. Second, as has been shown in question 1.a. users make fewer errors when using *CYNTHIA*. I answer this question with specific reference to the video experiment. The four subjects that took part in the videoing each worked through a maximum of three examples of increasing difficulty. Some of these were attempted using *CYNTHIA* and some without *CYNTHIA*. Table 8.7 gives the number of examples and timings for each student. Times are given in minutes and seconds. C denotes that the example was attempted using *CYNTHIA*. J denotes that the student did not use *CYNTHIA*. Note that all students were given a small amount of help. This is uniform except

for student 3 who was given a substantial amount of help. The examples involved writing simple recursive functions. Since the experiment was undertaken only two weeks after recursion had been introduced, the students understandably had some difficulty. Student 4 failed to finish `addlist`. Students 2 and 3 did not have time to attempt it.

Student	`leading0s`	`maxlist`	`addlist`
1	11:31 (C)	20.39 (J)	17:70 (C)
2	27:00 (J)	20:10 (C)	–
3	16:30 (C)	12:20 (J)	–
4	16:10 (J)	10:30 (C)	18:46 (J)

Table 8.7: Student Performance on Three Examples.

The general level of ability of the students seemed to be in the order: student 1 (best), student 4, student 2, student 3. `maxlist` is slightly harder than `leading0s`. `addlist` is more difficult because it involves multiple recursion. Students 1, 2 and 4 seem to have performed better with *CYNTHIA*. On the first two tasks, all of these students took less time when using *CYNTHIA* – on average, 35% less. Student 3 took longer when using *CYNTHIA* but the amount of help given was so great that a comparison is meaningless. A 35% gain would seem to agree with informal observations. For this level of task difficulty, the student often starts with a good idea of the required program behaviour and can describe this behaviour fairly accurately. Most time is taken up trying to implement this algorithm – for example, correcting syntax and type errors and perhaps adjusting their algorithm slightly. Hence, 35% represents the gain in implementation time achieved by *CYNTHIA*. Student 3 was unable to describe the algorithm in abstract terms and hence took more time in non-implementational work. This explains why the 35% decrease is not experienced in this case. All of these examples are relatively simple. An interesting question is whether *CYNTHIA* would provide a saving on more complex examples where implementation is more mixed in with trying to decide on the abstract algorithm required.

Clearly, too many conclusions should not be drawn from the timings given above. The number of subjects studied is very small and so can only be suggestive of a trend. It is unclear whether this perceived increase in productivity also appears in a natural setting. Certainly, it does not seem that SG2 progressed more quickly through the

course notes than SG1 despite the fact that they generally used *CYNTHIA* more. However, there are so many other competing factors that it would be very difficult to measure a change in productivity in such a setting. Further investigation is needed.

2. How does programming by analogy compare to writing a program from scratch?

This question is asked to compare the traditional approach of writing programs as a string of text to the transformational approach advocated in *CYNTHIA*.

2.a. Are the editing commands well-designed?

This question concerns the transformation of a source program using *CYNTHIA*'s collection of editing commands. Specifically, was the structure of the commands well-understood, was their function clear, etc.? Green et al [Green & Petre 96] introduce the notion of 'cognitive dimension', a broad-brush evaluation technique for interactive devices and non-interactive notations. Green describes thirteen high-level criteria for discussing the design of a system. The idea is that they will form a common point of discourse for evaluating interactive systems. Although I will not mention all of the dimensions here, they serve a useful framework for discussing the design of the editing commands and for evaluating how easily the editing commands can be learnt and applied. The following considers some of these dimensions and evaluates the set of editing commands on each.

Abstraction Gradient

Each editing command is essentially an abstraction, grouping together common sequences of editing operations. But are they at the right level of abstraction? As Green says, "learning to think in abstract terms is a high educational achievement". The natural question to ask therefore is if the students using *CYNTHIA* could understand the editing commands. Does the abstract nature of the editing commands benefit them in the long run?

The original aim when designing the editing commands was to make the set as small

as possible whilst keeping the meaning of the commands transparent to a new user. As far as the former goes, the goal was certainly achieved — with as few as 11 commands, a wide variety of programs can be produced (much wider than comparable systems such as [Brna & Good 96, Bundy *et al.* 91]). However, the high number of Incorrect Usage errors in Tables 8.4 and 8.5 show that the commands caused some confusion. As already mentioned, this is partly due to practical issues such as documentation. In addition, the abstractness of the editing commands seemed difficult to learn. Some commands are based on functional programming concepts — for example ADD ARGUMENT, MAKE PATTERN, CHANGE TYPE — and if the student has not got to grips with functional programming, he is likely to be baffled by the command. There is a chicken and egg situation here — learning the commands is easier if functional programming is understood, but use of the commands can help the understanding of functional concepts. See research question 4.a. for further discussion. Some students commented that they did not find the naming of the commands very intuitive. MAKE PATTERN is an example. Consider the following code:

```
fun combine x nil = nil
| combine [x] (h::t) = ...
```

To split the indicated `x`, students would try to rename `x` to `nil`. This would be disallowed by *CYNTHIA*. Or they would remove `x` and then use ADD ARGUMENT and enter `nil` for the new variable name. The problem here is that students do not think in terms of making a pattern or splitting a pattern or such like, but in terms of adding another line of code. One way to overcome this would be to recast the commands in terms of very obvious code-writing operations, such as ADD LINE OF CODE. However, we would lose something by doing this. MAKE PATTERN does more than just adding a line of code. There is a specific reason why the code is being added and MAKE PATTERN cannot be used to add just any line of code. I do not think that the command names should be reduced to the lowest common denominator.

Another common problem was failing to understand the distinction between RENAME and CHANGE TERM. The former is used to globally rename a variable whereas the latter changes an expression locally. Sometimes, students would try to change the result in a function definition by using RENAME.

It is unclear whether it is better to design the commands in terms of functional pro-

gramming concepts or not. The ideal solution is probably to incorporate the editing commands into the teaching material.

Closeness of Mapping

This dimension measures the closeness of the problem world and the program world. In this case, the problem world is ML programming and the program world is the set of editing commands. One issue that the course lecturer raised when integrating *CYNTHIA* into teaching was that there was a danger that the students would be learning *CYNTHIA*-ML rather than ML itself. This is true, but one has to consider the benefits of teaching ML in a single ten week course. The main reason for doing this is to show students that there are alternative ways of programming, the functional style being one of them. Hence, if the students learn *CYNTHIA*-ML, they are still learning functional concepts and can take these concepts away with them. In fact, the structure of the editing commands might serve to stress the functional concepts even more.

Consistency

This dimension asks: when some of the language has been learnt, how much of the rest can be inferred. In this context, the question concerns the consistency of the operation of the editing commands. It is instructive to look more closely at the Incorrect Usage errors committed by SG2: see Table 8.8 for a breakdown (the categories are explained in Appendix A.3). Four errors, U12, U14, U16 and U7, contribute over 60% of the

	Errors		Errors
U1	0	U11	4
U2	4	U12	12
U3	3	U13	1
U4	0	U14	11
U5	1	U15	4
U6	1	U16	11
U7	11	U17	1
U8	3	U18	2
U9	2	U19	1
U10	2		

Table 8.8: Incorrect Usage Errors.

total count. Three of these errors are in response to events that the user has every

right to assume should be possible. U12 says that MAKE PATTERN cannot be applied to integers (integers were dealt with in a sub-optimal way in the evaluation version. Because integers are not based on finite constructor definitions, it is non-trivial to allow the pattern edits to work with integers. The evaluation version required the user to start with an integer recursion if it was needed). U14 is similar — the result of CHANGE TYPE needs to introduce a pattern definition on integers. U16 is again something that *CYNTHIA* should support. U16 errors occur when the user fails to use ADD RECURSIVE CALL to introduce a new recursive call, but attempts to change an existing recursive call directly using CHANGE TERM. All of U12, U14 and U16 are areas where *CYNTHIA*'s functionality is lacking in some way. Note that the other kinds of errors — which crop up when the wrong parameters are introduced etc. — are very few in number. The remaining major error, U7, is also understandable — it is the confusion between RENAME and CHANGE TERM referred to earlier.

The four errors mentioned show a slight lack of consistency in the set of editing commands. Students have attempted operations that should be possible based on their knowledge so far of the commands. This should not be frowned upon too much, however, for each of the errors could be fixed relatively easily in a future version of *CYNTHIA*. Hence, the evaluation has pinpointed areas where *CYNTHIA* could be improved.

Hidden Dependencies

A hidden dependency is a relationship between two components such that one is dependent on the other but that dependency is not fully visible. An example would be a spreadsheet — a formula in a cell tells which other cells it takes its value from, but does not tell which other cells take their value from it. The first evaluation threw up an example of a hidden dependency in *CYNTHIA* to do with *CYNTHIA*'s type feedback mechanism. Sometimes a compiler will accept a function which is not accepted by *CYNTHIA*. Consider the `flatten` function:

```
fun flatten nil = nil
|   flatten (h::t) = h @ flatten t;
```

This is accepted by ML compilers and `'a list list -> 'a list` is inferred as the type for `flatten`. Unfortunately, certain interactions with *CYNTHIA* can lead to

the above definition but with an incorrect type displayed. Students would edit an old function, such as `doublist` which has type `int list -> int list`. They would correctly edit `doublist` into the `flatten` definition above. However, they would not edit the type declaration and so *CYNTHIA* would give a type error because the type declaration means that `h` is an integer and so cannot be appended onto `flatten t`.

This kind of situation was a cause of great confusion for students who encountered it. The students had an implicit assumption that the type declaration was correct and hence would scrutinise the program itself for errors. This implicit assumption is what makes type errors easier to locate in *CYNTHIA*. The situation given here, however, is an example where *CYNTHIA* introduces an error that ordinarily would not occur. Arguably, the student learns from the experience. The student is made more aware by solving the error that a nested list type is being utilised. This could have gone unnoticed using TEA.

One solution to the problem would be to highlight also the part of the top-level type declaration that is responsible for the error. In fact, this would be good even if the error is in the program — to help the user understand the error. More generally *CYNTHIA* should incorporate type inference so that it conforms more closely to the standard notion of programming in ML. See chapter 10 for more discussion on this.

Premature commitment

This dimension concerns the extent to which the user is forced to make a decision before the information is available. In the context here, this manifests itself as the degree to which the order of application of the editing commands matters. One of the main criticisms of the recursion editor [Bundy *et al.* 91], which is also a a transformation-based editor, is that the order of commands is critical to success and so the user must think about the order before delving into the programming task. To what extent is this also true of *CYNTHIA*?

For the most part, the order of editing commands in *CYNTHIA* is irrelevant. If the user applies an incorrect edit, and only realises this much later, it is easy to undo the edit by applying a short sequence of recovery commands. This contrasts starkly with the recursion editor, where the user can easily get stuck down an incorrect route from

which the only way to recover is to re-start from the beginning. In the worst case, the recovery commands in *CYNTHIA* will unpack those edits applied so that the user retraces his steps back to the start, but in the vast majority of cases, the recovery commands form a short route back onto the correct path.

There are a few examples, however, where *CYNTHIA* requires some form of premature commitment. In all of these cases, premature commitment merely makes life easier — it does not prevent the user from doing something. An example is where it is useful to decide upon the patterns that define a function initially, but where it can be awkward to revise this choice. Suppose the user is writing a function, `app`, to append two lists together and begins by splitting the second argument:

```
fun app l nil = l
|   app l (x::xs) = x :: app l xs;
```

It is at this point that the user realises he should have split the first argument instead. Ideally, there would be a command to transfer the patterns from the second to the first argument. Currently, however, the user must apply REMOVE PATTERN to give:

```
fun app l l2 = l;
```

and then MAKE PATTERN on `l`, giving:

```
fun app nil l2 = nil
|  app (x::xs) l2 = x::xs;
```

Note that the user must also apply CHANGE TERM in the second clause to re-introduce the program fragment that was lost during the application of REMOVE PATTERN[2]. Hence, although the user can achieve the desired goal, he needs to go a long way round to get there. Another example of this is when the user has initially chosen the wrong programming construct — suppose he introduced a conditional statement of the form `if (x:nat) = zero` into his program but now wants to use the `case` construct on `x` to introduce patterns for `x=zero` and `x= s n`. Although there are two cases for both `if` and `case`, there is no direct way to achieve this. The user must first remove the conditional statement then introduce `case` and probably re-implement parts of the program that had been deleted. This phenomenon is a consequence of the fact that

[2] One variation would be to allow the user to specify which clause is kept so that `x::app l xs` need not be re-typed.

the number of editing commands was deliberately kept to a minimum. More advanced users could easily find this sort of thing frustrating, however, and so commands such as one that transfers patterns from one argument to another would be of great use.

Observations of the students showed that occasionally, they found it easier to start from scratch rather than try to find the correct sequence of recovery commands. There are two cases when this can happen. The first is when the user applies an incorrect edit and cannot see how to correct it easily. The second is when the user gradually realises that he has chosen a sub-optimal starting example and so goes back and starts with a different example. There is sometimes a fine line between persevering with the current source example or choosing an alternative. Note that it can be a valuable learning experience to revise choices in this way. The student is refining their concept of similarity of functional programs. This will help them to develop an improved strategy for writing functions later (whether using *CYNTHIA* or not). This strategy is an essential part of an expert programmer's repertoire and hence, I claim that *CYNTHIA* is encouraging the novice user to think in this kind of way.

Progressive evaluation

Progressive evaluation means that programs can be evaluated by the user at frequent intervals during their development, not just once the program is completely finished. *CYNTHIA* improves on ML compilers in a significant way here. Although any program must be finished before it is executed (for it still must be accepted by the compiler), the user gets constant feedback about semantic errors during the programming process. This is achieved by the use of the highlighting mechanism for pointing out type errors etc. The key point is that *CYNTHIA*'s feedback merely notifies the user of a problem, it does not enforce them to change it immediately. Hence, the user retains the freedom to experiment but the existence of any errors is always in the back of his mind.

2.b. How easy is it to choose a starting (source) example?

Programming by analogy introduces two additional overheads for the user. The first of these is that of learning the transformations. This has been dealt with in question 2.a. Secondly, there is the decision about which source example should be chosen.

Some work has been done in the area of software re-use to provide sophisticated library systems that allow the user to quickly select the best example [Weber 96, Runciman & Toyn 91]. In *CYNTHIA*'s case, however, no library search functions are provided. This is for two reasons. First, since *CYNTHIA* is currently being used in a novice environment it was considered unnecessary. The students would be dealing with relatively easy examples and probably not building up too large a database of source functions. Second, the choice of a source is not as critical as in other systems because the editing commands are very flexible and so it is easy to recover from a sub-optimal choice. The evaluation phase gave an opportunity to test out these decisions.

One way of answering this question is to consider how students decided upon a source example. There seems to be three main ways — recency, familiarity and closeness. Most students pick the function they have used most recently[3]. For instance, if they have four functions to write, they use a pre-defined function as source for the first example. They then use the solution to the first task as source for the second, and so on. In many cases, this is a perfectly reasonable approach. For instance, when working on the on-line tutorial, examples within a tutorial tend to be similar (and get increasingly more complex) so that such an ordering is very natural. It is not quite so natural in a real situation. For example, in the crossover experiment, a student wrote `combine`, which has four definition clauses, then used `combine` as the starting point for a primitive list recursion example. Most students do modify their strategy in this sort of situation, however.

Another very common way of choosing a source is to choose a familiar example. In the version of *CYNTHIA* that students were given, 6 examples were pre-defined. Two of these were primitive list recursion examples, `sum` and `doublist`. These were familiar examples as they were used in the tutorial material on list recursion. Students were quick to pick one of these as a starting point rather than something they had defined themselves, even if their definitions were closer to what they needed. This is because the students are more familiar with the built-in functions and so need not waste time understanding them. It should be said, however, that because the nature of functional programming means that most functions are relatively small, understanding the source

[3] This backs up the claim made in [Weber 96].

example is rarely a time-consuming task. Also, students don't start to think about the task in hand until they have something on the canvas. Only once they have a function in the edit area do they start to think about the current task. By bringing something up in the edit area straight away, they feel as though they are part way to their goal.

The more able students do think more deeply about which source example to choose. This was brought out during the videoing. Student 1 said: "I'm looking for a function with two lines in it." when trying `leading1s`. Student 4 said, whilst looking at the definitions available: "So, I want to get something closest to `maxlist`. I don't know what half of these are unfortunately." He then selected a couple and decided they were not close enough until he eventually chose `sum`. The two main measures of similarity used in these circumstances were: the type of the variable being recursed upon, and how many patterns (or lines of code) were in the function. They did not seem concerned with the result type of the function or with the type of non-recursive input types. Student 1 looked for a function with 2 lines of code as source for `leading1s`. He chose `doublist` even though this has a result type of `int list` not the required `int`.

The course lecturer report does mention that some students had difficulty choosing source functions. My interactions with the students confirm this in that a few students would ask how they should go about making the choice. My feeling about this is that the students were trying to make the task more difficult than it was. Given that there was definitely a choice to make, these students were overly cautious and expected there to be some hidden set of rules for making these choices which they might be tested on later. It was as though the students could not believe that their initial choice was so unimportant and so they assumed they must be missing out on something.

The observed choice of source examples lends credence to the usefulness of the analogy approach. SG2 were specifically watched over the duration of the course to see if their choice of source example would change over time. The identity function

```
fun f x = x
```

was provided as one of the primitive *CYNTHIA* functions. By selecting this function, the students are in many ways ignoring the analogy approach. This never happened, however, suggesting that the students did find previous function definitions helpful.

In a more advanced, expert-oriented version of *CYNTHIA*, it might be useful to have some sort of library retrieval mechanism. The number of functions in the database would be much larger. A very simple, but potentially effective, solution would be to implement a library browser which annotated the function name with the type of the function and perhaps the kind of recursion scheme, or a sentence (provided by the user) of its meaning.

3.a. What was the attitude of the users?

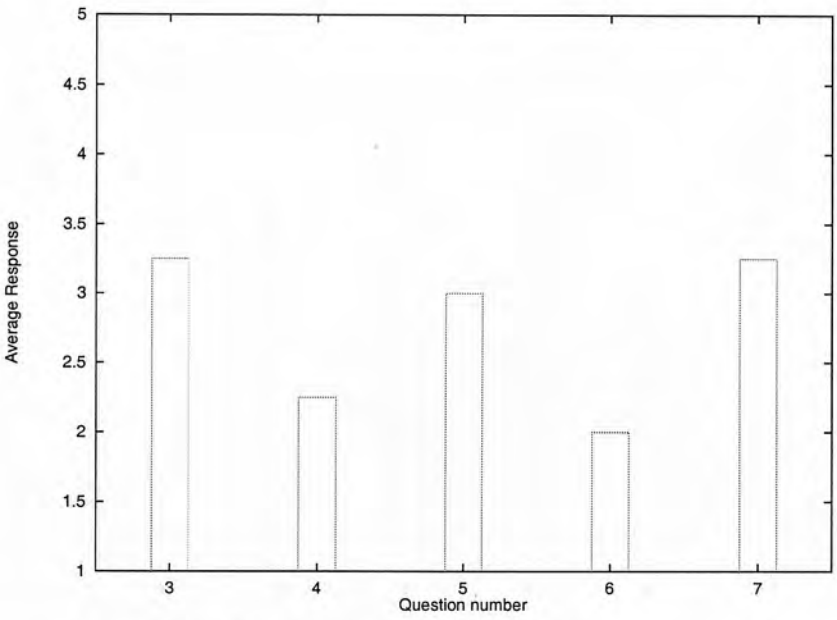
Both SG1 and SG2 were given questionnaires at the end of the course to assess their attitude towards *CYNTHIA*. These questionnaires are given in appendices A.1 and A.2 respectively. The questionnaires differ in content because the way that *CYNTHIA* was introduced was different for each group. Recall also, from §8.1.1, that the abilities of the two groups differed significantly. This seems to have had an effect on the results of the questionnaires.

16 questionnaires were returned from SG1. Figures 8.4, 8.5 and 8.6 give the results for questions 3–7 and 9–10.

Questions 9 and 10 ask about specific features of *CYNTHIA*. For these questions, the responses seem to be generally positive. Only 9(e) and 10(g) fall below the average score. 9(e) concerns the colouring of static semantic errors other than type errors (e.g. an unbound variable). These errors only occur rarely, so this is probably why the students did not find the feature overly helpful. Again, 10(g) is an error message concerning lack of termination of a definition. The students did not encounter this error frequently (only 4 out of the 16 students said they encountered it). However, it is expected that the students found non-terminating programs difficult to debug. Questions 3–7 ask about more generic features of *CYNTHIA*. 4–6 fall below the average. This slightly negative attitude is reflected even more so in SG2.

21 questionnaires were collected in total from SG2. The results of the questionnaires are given in Figures 8.7 and 8.8.

The results of the questionnaires are, in many ways, quite surprising. The students found ML slightly easier than other languages they had programmed in (2.67 score on

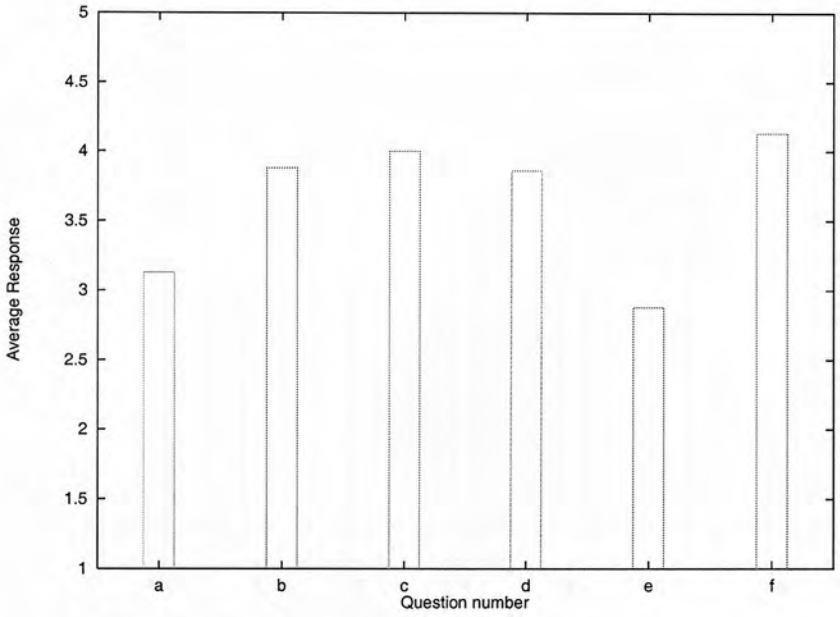


3 How useful is programming by analogy?
4 How difficult to transform start to the required program?
5 How efficient is writing programs in CYNTHIA?
6 How much slower would a text editor be?
7 How well are editing commands understood?

Figure 8.4: SG1 Questions 3–7.

average). And all of the various parts of the course scored very close to 3 points — see Figure 8.8 — suggesting that the students did not have any particular difficulties. Figure 8.7 shows a comparison between *CYNTHIA* and standard text editors. The students found it much easier just to use a text editor. They did, however, think that *CYNTHIA* was slightly better at providing help in locating type and syntax errors. The students were also asked whether they would prefer to use a standard text editor or *CYNTHIA*. Of the 18 students who answered this question, 17 said they would prefer a text editor. 1 said they would like to use *CYNTHIA* in the early parts of the course and then a text editor later on.

The attitude of the SG2 students is significantly more negative than those of SG1. It is interesting that, although students produce fewer errors using *CYNTHIA* (see research question 1.a.), they prefer to use a text editor. I believe that the students tended to base their answers more on the surface features of the systems, such as the



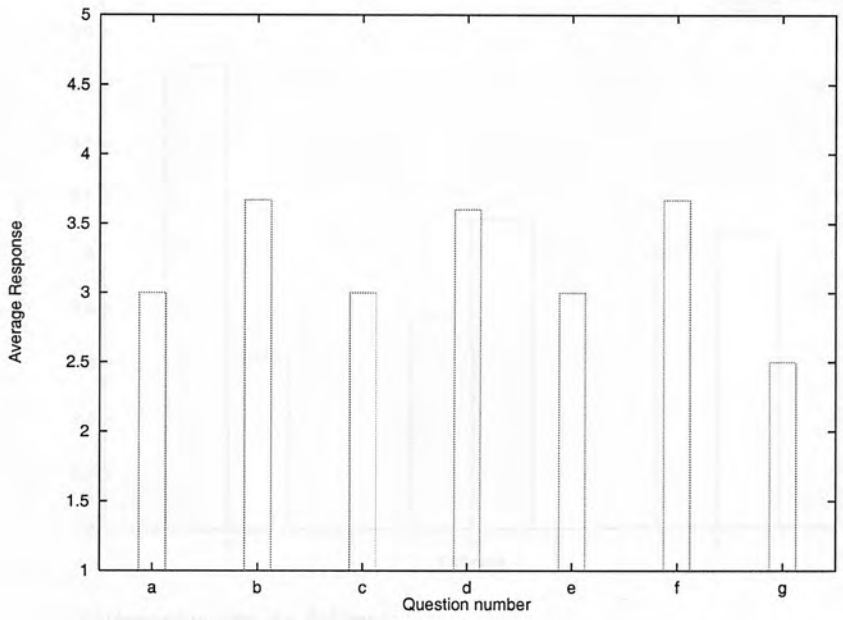
a) Less text typed in CYNTHIA
b) Syntax checking
c) Function type is visible
d) Inspecting types with right mouse button
e) Global updating (e.g. ADD ARGUMENT*)*

Figure 8.5: SG1 Question 9.

usability of the interface, rather than taking a more abstract view. All the open-ended questionnaires were answered by very low-level responses such as "change the colours of the highlighting" or "point and clicking can be slow at times". Hence, I think the apparent negative attitude of SG2 is more due to the fact that insufficient attention has been paid to the usability of *CYNTHIA*'s interface rather than the underlying ideas.

In addition to the questionnaires, four students agreed to take part in a "focus" group where they were asked general questions about the course and the systems used. This was meant to be an opportunity for very informal discussion. The responses served to back up the results of the questionnaires.

I now summarise possible reasons for negative attitude amongst students, based on conversations, the focus group and observations of individuals.

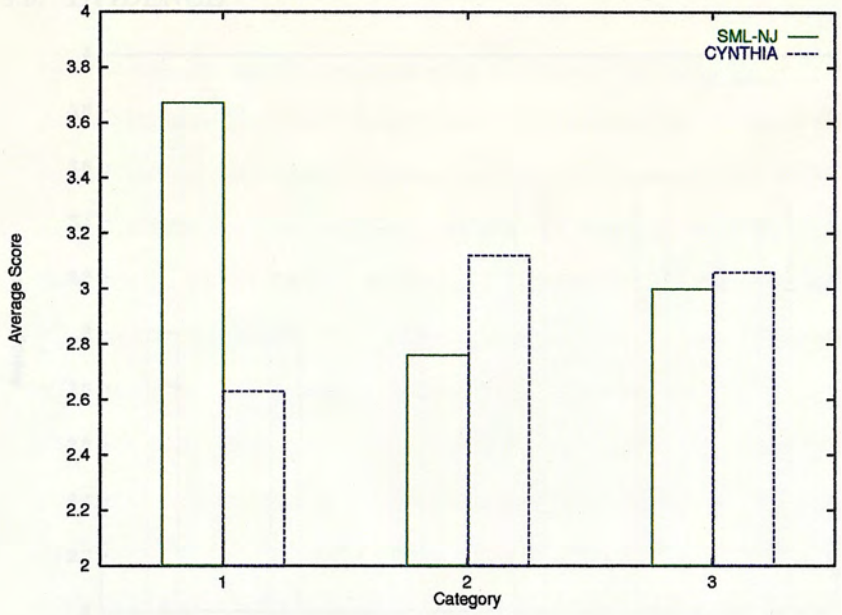


a) Syntax error
b) Wrong number arguments
c) Variable already in use
d) Type not valid
e) CHANGE TERM *can't alter recursive calls*
f) Pink highlighting
g) Can't show termination

Figure 8.6: SG1 Question 10.

Practical Issues

- The students found the system difficult to use. There are a number of usability problems that discouraged its use. See research question 3.b. for more discussion.
- *CYNTHIA* tended to have quite a slow response time. With 3–5 users on a single processor at Napier, the speed can be quite bad. It is by no means totally unacceptable, but it is bad enough to discourage use of *CYNTHIA*. I know of at least 1 or 2 students who gave up on *CYNTHIA* because of the slow responses.
- For SG1, there could have been more training given in the use of *CYNTHIA*. Students were just given an on-line tutorial to work through. Although extensive documentation and further examples are included on-line also, it was up to the students to go through it at their own pace. Hence, only a few of the keenest



Categories are as follows:
1 — How easy is the tool to use
2 — How helpful was it for identifying syntax errors
3 — How helpful was it for identifying type errors

Figure 8.7: *CYNTHIA* and SML-NJ comparison.

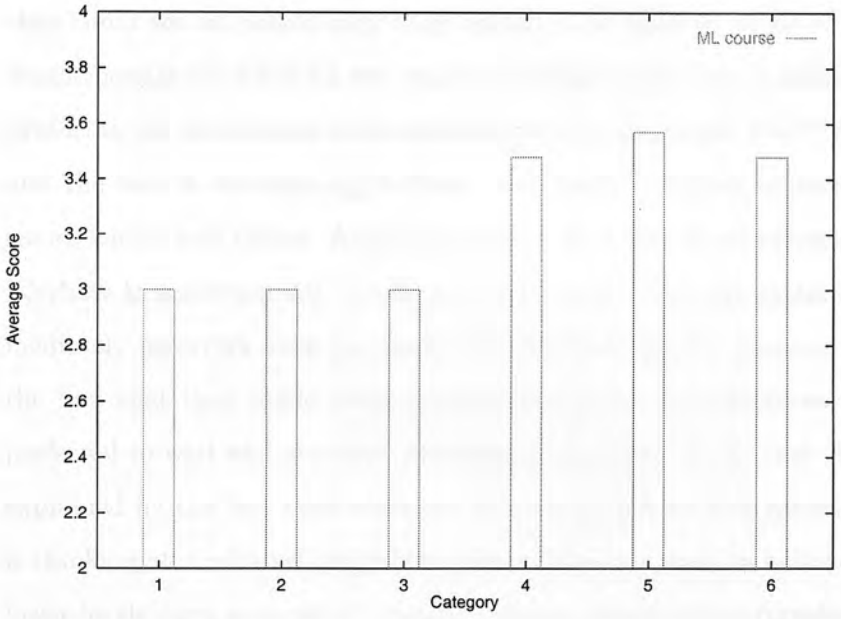
students bothered to look it up. This was rectified for SG2 by merging existing course notes with *CYNTHIA* documentation. See question 4.b.

- *CYNTHIA* does not support a large enough subset of ML for the course. *CYNTHIA* supported most of what was needed for the on-line tutorials but not everything. This was more a problem for SG1 since for SG2, the course was modified by removing parts that *CYNTHIA* cannot deal with.

Non-Practical Issues

Many students claimed that *CYNTHIA* was too restrictive. Here, they mean restrictive not in the sense that it supported an insufficiently large subset of ML, but in that it kept butting in with error messages when the user was trying to do something. As one student said in the focus group:

I liked the idea of just changing the bits that needed changing. But every time I tried to change something, CYNTHIA jumped in and said don't do



Categories are as follows:
1 — functional programming
2 — types
3 — recursion
4 — pattern matching
5 — user-defined types
6 — applications

Figure 8.8: Students Perceptions of the ML Course.

> *that, it will give you an error. I thought well, where do I start. I didn't build up enough intuition to decide where to start changing stuff. And 9 times out of 10 I would just hit a brick wall and so I got really frustrated with it.*

The students went on to say that understanding what each command did wasn't really the problem. It was more the fact that they would try to apply an editing command and *CYNTHIA* would prevent them from doing it because it would introduce an error. I think there are two situations where this happens. First, when the student enters something in a dialog box and it cannot be parsed. Second, and more confusing for the students, when the user is trying to apply an editing command to the wrong object. An example would be the U12, U14 and U16 errors discussed in question 1.a. *CYNTHIA* often prevented the student from doing something that in their minds

they could see no reason why they shouldn't be allowed to do it. The fact that students thought *CYNTHIA* too restrictive seems then to be a combination of usability problems, an incomplete understanding of the structures *CYNTHIA* operates upon and the way it operates upon them, and lastly, a feeling of lack of restriction that comes with a text editor. Anderson conducted some related research on his LISP tutor [Corbett & Anderson 92]. Anderson found that when the system gave feedback immediately as errors were produced, the students quickly became frustrated. Despite the fact that they made fewer errors under these circumstances, the students much preferred to wait and get error messages all together at the end. This can perhaps be explained by the fact that when the student makes his first attempt at a solution, he is thinking at a relatively high-level and so does not want to be brought down to much lower-levels such as syntactic considerations, which would interfere with his thought processes. Anderson concluded by saying that it is an open question whether the long-term gains are actually better or worse under immediate-feedback conditions. On the one hand, the students' thought processes do not get interrupted. On the other, they may spend a long time working with ill-formed objects without realising it.

3.b. Was the interface easy to use?

A lot has already been said to answer this question. Research question 1.a. showed that SG1 experienced a lot of usability errors when using *CYNTHIA*, but that these were reduced for SG2. Question 3.a. has shown that the opinion of the students was that the interface could have been designed better.

There are two main points to stress. First, students did not like the way of entering text — i.e. having to use the mouse *and* the keyboard. The lecturer report also points this out, saying:

> *As they grew in confidence the slightly clumsy interface became more of a hinderance. (By clumsy I simply mean that the mouse needs to be involved at all. Programs get written quicker without them).*

On the other hand, students did find it useful to have some structure editing facilities such as being able to highlight an expression based on its structure. What is needed is

an overhaul of the interface that would make *CYNTHIA* more of a hybrid between a text editor and a structure editor. Students could type as normal but movement of the mouse would cause expressions to be highlighted and clicking the mouse would produce pop-up menus so that editing commands could be employed. The exact interaction between the text and structure editing would be a major research goal in itself.

The second point to make is that *CYNTHIA* is presently lacking in practical facilities for sending programs defined in *CYNTHIA* to the compiler. Again, the lecturer report points this out. For example, although multiple definitions can be stored in *CYNTHIA*, the system saves each function to a separate file and so each function has to be loaded into the compiler individually. Further work could concentrate on integrating the compiler and *CYNTHIA* more so that functions could be sent directly to the compiler rather than going through intermediate file saving / file loading operations.

4. How useful was *CYNTHIA* as an addition to current teaching methods?

This question examines *CYNTHIA*'s role in teaching. Ultimately, *CYNTHIA* should be used by any ML programmer, but its particular use by novices raised some interesting issues.

4.a. What concepts were learnt that otherwise might not have been and were they picked up easily?

Although *CYNTHIA* was not designed primarily as a teaching tool and its scope potentially lies with the expert as much as with the novice, the fact that *CYNTHIA* was tested extensively by students makes one aware of its particular benefits in teaching. There are two main ways that *CYNTHIA* helps students to progress quickly.

First, it has been shown earlier in this chapter that *CYNTHIA* reduces the number of errors that students make. One of the great early barriers to learning any new language is that the user invariably spends large amounts of time debugging programs whilst he is getting used to the syntax. *CYNTHIA* reduces this time by reducing the number of syntax errors made. The same is true of other kinds of errors as well, particularly type

errors. By helping to eliminate the more trivial kinds of errors, *CYNTHIA* is freeing the student to concentrate on the more advanced features of the language. Hence, overall progress should be quicker. It is difficult to evaluate from the experiments whether this was actually the case. Although the video experiment provides tentative evidence of improvement over specific examples, it is impossible to tell if progress was effected over the course as a whole — certainly, it does not seem that any one particular group of students overall learnt more ML than any other. What does seem clear, however, is that for some students at least the level of frustration associated with the constant debugging of simple syntax or type errors is reduced. Unfortunately, there was also frustration associated with the correct operation of *CYNTHIA*'s editing commands, so the overall effect is blurred. In fact, comparisons between the two groups at Napier is unwise because of the very different initial characteristics of the groups.

As discussed in Chapter 2, functional programming, and ML in particular, contains some concepts that are difficult and foreign to imperative- and non-programmers. The typing system is different to languages such as C and Pascal. Recursion and pattern matching can be difficult to get to grips with. In some ways, *CYNTHIA* emphasises these concepts — for example, patterns are built up incrementally using the MAKE PATTERN command and must always be well-defined, and all recursive definitions are checked for termination. It is natural to ask two questions here. First, did the students find it useful to have their programs checked for termination and well-definedness (the case for well-typedness has already been dealt with in question 1.b.)? Second, did the students learn anything by having these concepts emphasised?

I will consider pattern matching and termination separately. Students generally have trouble forming non-standard patterns. Standard patterns such as `nil` and `h::t` are stressed in the course content and used so frequently that they present no problem. This is more because the student is quoting from memory, however, rather than understanding the fact that `nil` and `h::t` form a well-defined pattern. This becomes clear when students try more tricky examples. For instance, consider student 1 in the video experiment. Student 1 seemed the best of the four students used in the videoing yet still had severe problems with patterns. When attempting `maxlist`, using TEA, he tried to consider lists consisting of zero, one or more elements. Student 1 said:

"There's 2 cases. `h::t` when `t` is a list and `h::t` when `h` and `t` are two integers in a list. I want `t` to be a list ... I don't think that's clear to the compiler .... I can't say `h::j::nil` or maybe I can." Student 1 went on to produce the following:

```
fun maxlist nil = ...
|   maxlist t::nil = ...
|   maxlist h::j::nil = ...
|   maxlist h::j::t = ...
```

The last two clauses overlap. The fact that the student started with a grossly over-complicated set of patterns led to him taking a long time to come to a solution (and then a sub-optimal one). For this example, the student was using a traditional text editor. The desired patterns could easily have been produced by an application of MAKE PATTERN, although, of course, the student has to apply MAKE PATTERN in the right way.

So does the existence of MAKE PATTERN make things any easier? The students, particularly SG1, did find it difficult to understand how MAKE PATTERN works. It seems that because the command works on an abstract level — patterns rather than lines of code are being manipulated — the students needed to have a fair understanding of patterns to be able to successfully use it. Without this understanding, students were liable to apply MAKE PATTERN to the wrong sub-expression resulting in an unnecessarily complicated set of patterns or they would apply the command correctly but fail to realise why certain cases had been produced. SG2 were much better at using the command, but this again was probably down to their increased abilities and improved documentation.

The main advantage of MAKE PATTERN is that when used correctly it prevents students from missing out cases. There are also some minor advantages. Firstly, by using MAKE PATTERN, students encountered more complex pattern definitions than they otherwise would have done. Non-*CYNTHIA* users always tended to use the standard pattern even if a slightly more complex pattern was more appropriate. The users still produced correct answers but would often have partly constructive and partly (or wholly) destructive definitions which is not perhaps the best style. *CYNTHIA* was seen to encourage the use of more suitable pattern definitions. One example of this is as follows. First, define the queue datatype as follows:

```
datatype 'a queue = P | ++ of 'a * 'a queue;
```

One of the tutorial questions says: "Define the function `unfair` which takes two queues and returns a single queue with the first queue behind the second queue." The student had already defined:

```
fun     front P = raise excep
|       front(x++P) = x
|       front(x++y++z) = front (y++z);

fun     remove P = P
|       remove(x++P) = P
|       remove(x++y++z) = x++remove (y++z);
```

The student attempted to write `unfair` (using *CYNTHIA*) in a destructor-style fashion:

```
fun unfair p q = if nullq p then q else p;
```

The student then tried to change the `else` result to

```
unfair (remove p, (front p) ++ q)
```

This is in fact wrong, but you can see what the student is trying to do. He is trying to write a destructor version of `unfair`. *CYNTHIA* is not very adept at dealing with recursion in destructor definitions. This is because recursion is based around the induction scheme, which is in turned based around constructor definitions. In this particular example, it makes more sense to define the function constructively. A destructor definition using `if then else` might lead to the student missing out a case. In addition, it requires the definition of three auxiliary functions: `nullq` and functions to return the head and tail of a queue. There are, however, cases where destructor definitions are useful and they should be possible in an expert version of *CYNTHIA*.

There is, of course, a toss up here between restricting the user to a particular programming style and risk infuriating him because he cannot do exactly what he wants. My feeling is that the former cuts down debugging time in the long run and leads to clearer programs. See research question 3.a. for more discussion about this.

The other issue concerning patterns is the well-definedness. The fact that *CYNTHIA* insists on well-definedness means that sometimes programs can be overly long — since cases have to be included even if they are never going to be used. As far as novices are

concerned, this seems to be a good thing — I saw cases where it prevented students omitting cases accidentally. As far as experts go, however, it would probably be too cumbersome. However, one simple extension of *CYNTHIA* would be to provide the facility for the user to interactively suppress cases — MAKE PATTERN would initially yield all cases then the user could remove them selectively with a click of the mouse. Note also that the underlying proof would retain each case but the deletion would come when translating from proof to program. By doing this, there is little possibility that the user omits a case *accidentally* — it has to be the result of a conscious decision.

Let us come to the issue of termination. The students rarely wrote non-terminating functions. Only a small number of places were found in the scripts when a user attempted to introduce a recursive call with a non-decreasing measure and all of these were of the form `length (h::t) = ...length (h::t)...`[4]. In many ways, this is not surprising. The vast majority of programs students write are primitive recursive and therefore well-understood. Of course, the fact that users modify a *CYNTHIA* template means that they are less likely to make clerical errors to introduce non-termination into even primitive recursive functions, but I have no data to explain how much of a saving this is. Even then, however, one might ask if a technique as complicated as Walther Recursion is really needed. It may be better to use a heuristic approach that checks for commonly occurring termination errors such as the one given above. Whether a termination checker would be more useful in more everyday programming is unclear. Certainly, the students in this experiment attempted relatively simple examples only. If they progressed to more difficult kinds of recursion, termination may have been more important. Note, also, that the students did not encounter examples that required reducer / conserver lemmas to derive termination.

Another point is that potential termination errors may simply not show up in the *CYNTHIA* scripts, because the user is following a schema-based approach. For instance, the following function is non-terminating:

```
fun half x = if x=0 then 0 else 1 + half (x-2);
```

In *CYNTHIA*, the user is unlikely to even try to write a function in this way. Instead,

[4] Note, that I did not count functions over integers that do not terminate for negative arguments, since this kind of function was provided as a starting point, valid as long as only positive arguments are considered.

he would start with a function like the following[5]:

```
fun t 0 = ...
|   t 1 = ...
|   t n = ... t (n-2) ...
```

Although this still loops for negative integers, it terminates for positive numbers at least. The fact that the user is starting with a sound function means termination errors are less likely to occur. I think that non-terminating programs are more likely to be of this kind — i.e. over integers — than over structural datatypes such as lists.

It is possible that a general-purpose termination checker would be more useful in other languages. Haskell, for instance, may benefit from a checker that prevents the usual length function being applied to infinite lists. This would be a different sort of termination checker from Walther Recursion. Prolog is perhaps a good choice. Termination errors are more likely to occur in Prolog because of things like backtracking. However, as noted in [Bundy *et al.* 91]:

> *non-terminating and/or ill-defined, but none-the-less useful procedures are more common in Prolog than any other programming language I can think of. Hence, the editor I propose here would probably be more useful for other recursive programming languages than it is for Prolog.*

The recursion editor [Bundy *et al.* 91] only dealt with structural recursions — other termination checking techniques would be required to deal with non-structural recursions in Prolog.

One of the unexpected advantages of having termination checking in *CYNTHIA* is that it seemed to have an effect on the way that the course at Napier was taught. The lecturer of the course stressed termination more than usual (see his report in Appendix A.6). In fact, given that the course was a Formal Methods rather than a programming course, the lecturer introduced *CYNTHIA* to the students as an example of Formal Methods in practice. This, of course, helped to motivate the students in studying Formal Methods and helped to emphasise the significance of the concept of termination.

[5] A function similar to this is one of the primitive starting functions provided in *CYNTHIA*.

4.b. Was programming style affected?

It seems as though *CYNTHIA* does encourage a particular kind of programming style. In some cases, *CYNTHIA* restricts the user to a technique. In other cases, *CYNTHIA* is more suggestive rather than restrictive. One example is the colouring of ill-typed expressions. As described in Chapter 6, these expressions can be used as "stepping stones" to guide program development but need not be corrected immediately. This is certainly the behaviour observed by the students. Another example is where *CYNTHIA* encourages a top-down approach to programming (within individual function definitions). This tends to be a very good way to start programming in functional languages. Many approaches to teaching functional languages concentrate on getting the student to identify the correct type first and then the patterns to define the function and the recursion scheme[6]. *CYNTHIA* is a framework in which this order becomes very natural. First, each function must be given a type declaration and hence the students are much more conscious about the type of their definition. For example, student 1 when writing `leading1s` using *CYNTHIA* said : "I want it to take `int list` and want it to return an integer, so I'm going to change this." When using a text editor, he verbalises nothing about the type of `maxlist` which he is defining. He launches straight in, typing

```
fun maxlist nil = nil
```

not realizing that `nil` is not the correct type here. *CYNTHIA* makes the student think about the type because the type is displayed at all times and because they change the source function type to fit the next task. This means both that students are less likely to carelessly introduce type errors and that they become more adept at dealing with types.

The structure of the editing commands also suggests a top-down approach. Close observation of students shows a consistent order of application of commands. First RENAME is used to change the function name. Next CHANGE TYPE is used to modify the type declaration and then MAKE PATTERN and ADD RECURSIVE CALL are used if necessary. After this, ADD CONSTRUCT and CHANGE TERM are applied. Of course, this kind of structured approach appears with a standard text editor as well — students

[6] personal communication with Colin Runciman

often write a base case and then step cases because they are aware of the necessary structure for recursion — but *CYNTHIA* tends to emphasise the idea. This is particularly useful in the early stages of programming or for weaker students who take time to master the top-down approach.

Finally, the course lecturer mentioned one thing that he had noticed among the students. Students become more aware that expressions in ML are structured objects not just plain text. Students generally seem to have quite a lot of difficulty grasping this. *CYNTHIA* enforces it because of its highlighting mechanism. When the mouse is placed over a symbol, the expression with that symbol as functor is highlighted. For example, in `x + (3 * y)`, placing the mouse over `*` highlights `3 * y` and placing the mouse over `+` highlights the entire expression. This tells the student both that they are dealing with a structured object and also in which order the object will be evaluated.

4.c. Which students benefitted most from the use of *CYNTHIA*?

It seems as though weaker students got more out of *CYNTHIA* than strong students. This was observed in the individual evaluation groups — the stronger students would get frustrated more easily by the point-and-click interface. Groups SG1 and SG2, as described in §8.1.1, were of significantly different abilities. The weaker group, SG1, seemed to like *CYNTHIA* more. Having said that, both groups committed fewer errors when using *CYNTHIA*.

I believe that the frustration of the stronger students was mainly caused by practical issues — such as the point-and-click interface being too slow or the response of the machines being too slow. If the interface was re-designed as a hybrid editor (see research question 3.b.) these problems might be alleviated. Further research would be needed.

4.d. How easily was *CYNTHIA* incorporated into existing teaching methods?

The original intention when designing *CYNTHIA* was that it would require no extra training to learn the editing commands and general operation. As a result, the only documentation for SG1 was very limited on-line documentation, a half-hour tutorial

and a Web page describing the functionality of each command. In hindsight, this Web page was more suitable for reference purposes and as a result it was not referred to as much as it could have been. The editing commands do need to be learnt and provision should be made for this either in lectures / course material or by improving documentation. This was done for SG2 by merging the *CYNTHIA* and Napier documentation and it seemed to be more successful, although not entirely. I think the best approach would be to introduce the editing commands into the lecture content. As each editing command is encountered, a short motivation and description could be given. This would help the students to remember the commands — it was noted that students understood the explanations of commands in the documentation but would forget functionality between tutorial sessions. Note that *CYNTHIA* should not of course be dependent on lecture material as then it would be specific to a particular course. In practice, no system is really lecture independent — instructions on operating a tool are usually given in the early days of lecturing and it is often up to course organisers or computer support officers to give help on the use of systems because students lack the motivation to look up the information for themselves. Indeed, in the course at Napier, *CYNTHIA* was at a distinct disadvantage because considerable amount of instruction on text editors and the compiler was given throughout the course. This was not done for *CYNTHIA* as it was unreasonable to expect the computer support officers to become acquainted with the system.

From a practical point of view, it was relatively easy to incorporate *CYNTHIA* into the existing course structure. Apart from the documentation issue, the main changes made to the course were to remove parts that depended on syntax or concepts that *CYNTHIA* does not yet support — namely, tuples, op, real numbers. Quoting from the course lecturer's report:

> *Cynthia threw up several issues to be discussed in lectures. Types, termination, GUI's. Mostly I was impressed by how little the teaching material needed to be adjusted.*

There seem to be a number of opportunities for exploiting features of *CYNTHIA* in teaching. In this way, *CYNTHIA* would be used interactively to help more directly in

tutoring and could give students immediate feedback. Some examples are:

- Students could be asked to ascertain if expressions are well-typed or not. They could check their answers by entering them in *CYNTHIA*. They could then be asked to predict what happens to the highlighting if they change certain expressions. Note that this sort of thing is often done using traditional methods. However, I see two advantages to using *CYNTHIA*. First, the approach is more interactive. Rather than getting a type error message or a teacher's pre-written piece of text as feedback, the program reacts immediately to the student by changing colour. This will speed up the process of understanding and hence the student could get through more examples. Second, the complexity of the examples could be much greater. In course notes, very simple examples are usually presented so as not to swamp the reader. Using *CYNTHIA*, the user could build up complex examples for themselves. They would have a great deal of flexibility to experiment.

- As noted above, termination checkers seem to have only a limited benefit in program writing. However, it would be useful to see a checker in action when learning about the different types of recursion. Students could be taken through a variety of different recursion schemes, at each stage interacting with *CYNTHIA*, and noting *CYNTHIA*'s feedback. If *CYNTHIA* was equipped with termination explanation facilities, the student would get constant explanations why a definition was or was not terminating.

- As described in question 2.a., students have difficulty with patterns and deciding if they are well-defined or not. There are opportunities for interactive exercises using MAKE PATTERN or CHANGE TYPE. It strikes me that the incremental way of building up patterns, as is done when using MAKE PATTERN, is a very good way of teaching patterns. It is easy to see whether a set of patterns is well-defined by looking if it could have been produced using MAKE PATTERN (this does not need to have any relation to *CYNTHIA* at all, in fact). This, I believe, is better than the alternative of trying to look for cases that are not covered by the patterns.

8.7 Summary of Findings

This section repeats the main findings from the two evaluations.

- Students make fewer errors when using *CYNTHIA* as opposed to TEA.
- Once errors are made, they are easier to locate using *CYNTHIA*.
- Structure editing should be replaced by hybrid editing.
- Further investigation is needed to assess the worth of advanced features such as checking termination and well-definedness.
- Further effort should be placed on the design and introduction of the editing commands. The current design of the editing commands are sufficient but could be improved upon.
- Practical aspects of *CYNTHIA*, such as response speed, could be improved upon.

Mavis woke the next morning with a start. She was sure she had heard something. Anyway, not being able to hear it now, she assembled her things and decided to set out for the castle. No sooner had she done so, though, when she was confronted with a slight problem. In front of her, about ten paces away, a pack of angry-looking wolves lay in wait. One of them snarled, "So you've come for knowledges, have you? We'll see about that. You need teaching a lesson. We don't like people coming here and bringing order to the forest. We don't like order. It's far nicer when everything's chaos, then we can sneak up behind people and..." At this point, the wolf jumped towards Mavis. Mavis was so taken by surprise that the wolf got her backpack. It tossed it aside and the other wolves attacked it, ripping it open and throwing the contents all over the place.

Mavis ran back the way she had come. But she couldn't go very far for the wolves had moved around to encircle her whilst she had been distracted. "Oh dear," thought Mavis. There were wolves, on every side of her, slowly and menacingly walking towards her. Mavis screamed. But there was no-one to hear it. The wolves continued to advance and were almost upon her.

"Away with you," came a loud cry from somewhere behind Mavis. She turned and looked up to see a huge, strange looking bird descending towards the wolves, firing bolts of fire that scorched the wolves' toes. The wolves turned and fled. As the majestic bird drew nearer, Mavis noticed two figures on its back. They were riding the bird like a horse. "Hello Mavis," came a greeting. And as the bird landed, Mavis realised who it was. It was Cynthia and Walter. "Got here in the nick of time," said Walter.

"What is that?" asked Mavis, pointing to the bird. "Oh that," dismissed Walter. "That's a pecky-feed-bat!"

Chapter 9

Related Work

This chapter compares *CYNTHIA* to other novel editing environments. A number of approaches have been developed as frameworks for language-based editors. The most notable of these are attribute grammars [Alblas & Melichar 91] and CENTAUR [Borras *et al.* 88]. The Synthesizer Generator [Reps & Teitelbaum 89] uses attribute grammars to keep track of semantic information during the editing process. CENTAUR allows the user to express the semantics of a language in a Natural Deduction style. In the case of CENTAUR, an editor for ML has been developed which shares some of *CYNTHIA*'s features. Editors developed using attribute grammars and CENTAUR are structure editors that provide the guarantee of syntactic correctness and make some analysis of semantics. The only editor, apart from *CYNTHIA*, that guarantees termination is the recursion editor [Bundy *et al.* 91]. In many ways, *CYNTHIA* grew out of the work on the recursion editor and was designed to remove many of its deficiencies.

9.1 Attribute Grammars

This section describes the attribute grammar framework for defining language-based editors. A comparison with proofs-as-program is made.

An attribute grammar (AG) is a context-free grammar extended by attaching *attributes* to the nonterminal symbols of the grammar and by supplying *attribute equations* to define attribute values. Attribute grammars provide a powerful mechanism for ex-

pressing dependencies within a tree. They have been used as a technique for defining structure editors for programming languages [Reps & Teitelbaum 89] that also have some knowledge of the semantics of the language. Hence, AGs provide an alternative framework for defining an editor such as *CYNTHIA*. This section explains the advantages of proofs-as-programs over attribute grammars.

First, I give a description of attribute grammars. Given a production $p : X_0 \rightarrow X_1 \ldots X_k$, each X_i $(0 \leq i \leq k)$ denotes an *occurrence* of a grammar symbol. Each nonterminal occurrence has an associated set of *attribute occurrences* corresponding to the nonterminal's attributes. Each production has a set of attribute equations which define the production's attributes in terms of other attribute occurrences in the production. A program may be represented as a tree that is *consistently attributed* according to the attribute equations. When a tree is modified by editing operations, some of the attributes may no longer have consistent values. Hence, the tree needs to be re-evaluated to make the attributes consistent. [Reps & Teitelbaum 89] presents an *incremental attribute evaluator* which produces a consistent, fully attributed tree after each editing operation, minimizing work by confining the scope of the re-evaluation.

The Synthesizer Generator (SG) [Reps & Teitelbaum 89] is a system for writing attribute grammars which can then be used to generate a language-based editor. A specification of such an editor consists of the following components:

- A grammar describing the abstract syntax of the language. This is the basis for the structure editing facilities.
- Attribute equations. These encode information about the semantics of the language.
- An unparsing scheme (expressed as an AG), describing the display format, plus which objects are directly editable as text, etc.
- A grammar describing the concrete syntax of the language, along with a translation from concrete to abstract syntax.
- (optional) A description of transformations that may be performed on parts of the abstract syntax tree representation of a program.

The best way to make the comparison between AGs and proofs-as-programs is to discuss how *CYNTHIA* might be implemented using AGs, and in particular using the Synthesizer Generator. Figure 9.1 gives a grammar describing a possible abstract syntax for a subset of ML supported by *CYNTHIA*, including attributes for propagating types. Attribute equations are contained within the braces. For a symbol, S, $S\$n$ refers to the nth occurrence of S, where the first occurrence is on the left-hand side of the definition. If t is an attribute, then $S\$n.t$ is the value of the t attribute for the nth occurrence of symbol S. `ResultType` is a function to return the result type of a function type. `Lookup` is a function explained later. Note that Figure 9.1 only gives equations for propagating attribute information. In a real system, there would be additional equations for maintaining consistency. Note also that Figure 9.1 is only meant to be illustrative, not a working implementation.

The above grammar has been deliberately designed to try and capture the structure of ML programs in the same way that proofs-as-programs does (see below for more about this). One point to note is that attribute grammars are much more general than the proofs-as-programs approach. It is perfectly feasible to reproduce the proofs-as-programs idea using attribute grammars — e.g. to implement *Oyster* in the Synthesizer Generator. In what follows, I assume this is not the approach taken (it would probably be very inefficient), but that *CYNTHIA* would be implemented more directly. One nice fact about the SG is that everything (apart from auxiliary function definitions) is written as an attribute grammar. This contrasts with *CYNTHIA*, where the proof machinery is implemented in *Oyster*, the interface is written in Tcl/Tk, the parser is written using Yacc, etc. Although this is conceptually nice, it does tend to mean that in complex applications such as *CYNTHIA*, the AG database quickly becomes very large and convoluted, making it difficult to maintain [Paakki 95]. Standard attribute grammars were developed in the 60s and since then a number of extensions have been developed to deal with various deficiencies. These problems have been overcome to some extent with the introduction of modular and object-oriented AGs.

I will now present three main arguments why I think proofs-as-programs has something to offer over AGs.

```
clauseList  :  ClauseListPair(clause clauseList)
                          {clauseList$1.type = clause$1.type;}
            |  ClauseListNil() {clauseList$1.type = EmptyType;}
            ;

   clause   :  Clause(pattList witness)
                {clause$1.type = pattList$1.type -> witness$1.type;}
            ;

   witness  :  If(witness witness witness)
                             {witness$1.type = witness$3.type;}
               Wit(exp) {witness$1.type = exp$1.type;}
            ;

   exp      :  Appl(fexp expList)
                       {exp$1.type = ResultType(fexp$1.type);}
            |  Identifier(ident)  {exp$1.type = Lookup(ident,type);}
            ;

   pattList :  PattListPair(patt pattList)
                  {pattList$1.type = patt$1.type * pattList$2.type;}
            |  PattListSingle(patt) {pattList$1.type = patt$1.type;}
            ;

   patt     :  VPatt(var)      {patt$1.type = Lookup(var,type);}
            |  CPatt(constr)   {patt$1.type = Lookup(constr,type);}
            |  ApplPatt(constr pattList)
                    {patt$1.type = ResultType(Lookup(constr,type));}
            ;
```

Figure 9.1: Abstract Syntax for a Sublanguage of ML.

9.1.1 The Design Argument

Level of Abstraction

It seems to me that AGs are fundamentally linked to the grammar of the language in question which means that everything is done at a finer level of granularity. The main way in the SG for broadening this granularity is via the abstract syntax grammar. It can easily be seen that Figure 9.1 is at a much more abstract level than the concrete grammar given in the SML Definition [Milner *et al.* 90]. Even so, proofs-as-programs is more abstract still and, more importantly, is sensitive to the structure inherent in functional programming. Recall from Chapter 5 that the Curry-Howard

correspondence relates functional programming and synthesis proofs. The structure of these proofs is very similar to that of functional programs. AGs are just as suitable for non-functional languages. In fact, the proofs-as-programs idea can be applied to non-functional languages, as well, but a bit more work has to be done identifying the correspondence.

At a design level, therefore, proofs-as-programs is more suitable as the basis of an ML editor. One example of this is the use of an induction scheme which localises all information regarding the termination of the definition. This localisation lends itself easily to analysis, whereas, although it is easy to imagine termination analysis within AGs, the grammar would probably become complex and unreadable.

Consider, too, the effect on the editing commands. MAKE PATTERN in proofs-as-programs merely corresponds to a change of induction scheme. In contrast, in using the abstract syntax representation given in Figure 9.1, changes would be not be localised and so would be needed in both `clauseList` and `patt`. The structure of the synthesis proof directly mirrors the structure of the ML program as well as the structure of the editing commands. In an AG framework, it is difficult to see how anything more than a good approximation could be achieved.

There is a practical impact involved too. Because of the lack of localisation, attribute values have to be propagated over the grammar. This can easily be done — this is after all one of the strengths of AGs — but there are a number of disadvantages to this. First, as noted by [Paakki 95], this requirement tends to mean that a lot of attribute equations are "transfer rules" of the form $X.a = Y.a$ which merely duplicate attribute values. It would be easy for the developer to miss out one of these. Second, a lot of attribute equations would be in some sense "redundant" because they attach attribute values to syntactic categories for which the attribute makes no sense conceptually. For example, imagine trying to check that the recursive calls in a function definition are measure decreasing. This requires the definition of an attribute, `measure`, to keep track of the measure of each recursive call. Then, at some point two `measure` attributes must be compared. However, to link up the recursive calls, the `measure` attribute values must be passed through intermediate nodes, where it makes no sense to have a value for `measure`. This confuses the issue of which attribute equations should be defined at

which nodes and hence is a potential source of bugs.

Editing Commands

I think proofs-as-programs, equipped with suitably defined general tactics, is a much better framework for implementing *CYNTHIA*-like program transformations. The key point here is again the more abstract level of the synthesis proof. Imagine implementing a global editing command such as ADD ARGUMENT. Given a general tactic that applies I-$\forall$ (Figure 5.1, p. 92) repeatedly, all that needs to be done in the proof framework to realise ADD ARGUMENT is to add the new type to the specification. The effect of this is then automatically propagated by just replaying the original proof tactics. No change is needed to any other part of the synthesis proof. Although AGs are good at propagating effects in this way, this propagation is only done attribute by attribute. Hence, if CHANGE TYPE was used to change the result type of the program, the AG would propagate this effect to the leaf nodes of the abstract syntax tree and flag type errors, just as *CYNTHIA* would. However, when coming to implement ADD ARGUMENT, the developer would find that there was no attribute available for propagating a change such as this — since each occurrence of the function being defined must be modified. Hence, the developer has to introduce a new set of attribute equations to deal with this, and again, this is a potential source of bugs.

It has been recognised that standard AGs are not ideal for implementing program transformations — in the CACHET program transformation system [Liu 95] based on the SG, a metalanguage for complex tree transformations was added. This was a non-trivial extension. Local transformations, such as replacing a piece of syntax by a template, are relatively easy to perform, but global operations either require the implementation of a generic mechanism or the definition of new sets of attributes.

Error Reporting

The key point here is the fact that proofs-as-programs is more geared to detecting and giving feedback about programming errors. Static semantic errors such as an unbound variable are detected naturally when the variable fails to appear in the hypotheses list. Although all of this can be done using attributes, each error needs its own attribute

which is propagated through the abstract syntax tree. When considering an example as complex as *CYNTHIA*, the collection of attribute equations would quickly become complex as large amounts of information have to be passed around the parse tree. AGs can overcome this somewhat by the definition of an attribute `env` which would essentially be the hypothesis list. This means that only one attribute `env` has to be propagated rather than lots of them. Even so, each kind of error must be checked individually, whereas the framework is more general in *CYNTHIA* — since an error just corresponds to a failed rule application.

9.1.2 The Globality Argument

Although attribute grammars are good at propagating attributes, i.e. globally analysing program trees, there are some problems. It has already been mentioned that a large number of transfer rules are often needed. Some systems have implemented special mechanisms for overcoming this. The Syntheziser Generator has *upward remote attribute sets.* These are special sets of attributes which reside at the root of the grammar and can be accessed directly by a lookup function from anywhere. An example is given in Figure 9.1 where `Lookup` is used to find out the type of a variable from the specification rather than having to include transfer rules to access the root. This introduces a notion of globality into the SG that otherwise would be awkward to achieve. Other systems provide the transfer rules as defaults.

It is not always possible to make use of upward remote attributes as they always reside at the root of the program tree. In many cases, attribute values need to be passed between two non-root nodes. An example is when trying to derive conserver lemmas. This requires that two attribute values are compared at some point within the tree (we are comparing two measures). Hence, the attribute values must be passed through intermediate nodes, and as with the recursive call case earlier, `measure` may not make sense for some of these nodes.

Hence, although AGs are fine for global analysis, the solutions can sometimes be awkward, lengthy and difficult to debug.

9.1.3 The Correctness Argument

Another major point is in considering the correctness of an implementation of *CYNTHIA*. By using proofs-as-programs, it is feasible to make a claim that all programs written in *CYNTHIA* are correct (with respect to a weak specification). Since *Oyster* is an established proof checker, we have confidence that its inference rules are correct. Hence, since a *CYNTHIA* program is a proof made up of these inference rules, it must also be correct — otherwise, the rules would not apply. A word of warning here. For reasons of efficiency, rather than using tactics, *CYNTHIA* uses a set of newly-defined inference rules and the soundness of these rules could be open to question. Still, though, it is much easier to check the correctness of these rules than in an AG-based *CYNTHIA* because *Oyster*'s mechanisms for applying and combining rules are still used.

I have already hinted at reasons why an AG-based *CYNTHIA* would have serious difficulties in providing comparable guarantees of correctness. Ultimately, a verification proof of the attribute grammar would be required. But, even stopping short of this, it is not clear that the program developer would have much confidence in the correctness of his implementation. In the case that the transfer rules have to be written by hand, it would be easy to omit one. As mentioned previously, attribute equations have to be given at nodes where it makes little sense to define them (cf. the `measure` example, given earlier). Hence, it would be understandable if the developer missed out such equations, thinking that they were not needed. Finally, the nature and complexity of the attribute equations mean that it is difficult to keep track of what is going on. Key attribute values (such as those representing the measure of two expressions to be compared) may be widely separated in the attribute grammar, making it difficult for the developer to maintain the salient points in mind.

A further word of caution is needed in order to not over-exaggerate the guarantees that *CYNTHIA* does provide. *CYNTHIA* does contain a substantial amount of arbitrary code that could be flawed, most notably the Tcl/Tk code used to implement the interface. This can mean, for example, that although the underlying synthesis proof is correct, it has been displayed incorrectly to the user so that an incorrect program

actually appears — in contradiction to *CYNTHIA*'s claims. Note, however, that this is a relatively easy problem to overcome. The solution would be to have two separate mechanisms for translating the synthesis proof. The first would display the proof to the user using Tcl/Tk code (as is done at present). The second would be a much simpler transformation of the proof into text. It is this version that would be saved to a file and compiled. This latter translation would not involve Tcl/Tk code or code for highlighting inconsistent expressions. The much simpler nature of the second translation would mean that one could be more confident of its correctness.

Proofs-as-programs is also more suitable for the kind of analysis (e.g. termination analysis) that is carried out in *CYNTHIA*. The proof framework is a recognised representation for this sort of analysis and hence, the path from research paper to implementation is a short one.

So far, I have concentrated on where proofs-as-programs wins over attribute grammars. There are, however, some areas where proofs-as-programs is weak. The principal deficiency is efficiency. Replaying inference rules in *Oyster* is notoriously inefficient. Hence, although the replay of rules in *CYNTHIA* can sometimes be slow, it would be improved considerably if efficiency was taken into account more. Certainly, the efficiency of the rule applications could be improved fairly easily. The implementation of *Oyster* is well-known to be inefficient. Other theorem provers tend to be more efficient at applying inference rules.

At present in *CYNTHIA*, only one function can be edited at once. This means that only one synthesis proof needs to be replayed after each editing command. Updating attributes in the AG framework is traditionally also inefficient — although more because entire programs rather than single functions are replayed. A lot of work has been done to make AG's more efficient, notably Thomas Reps's incremental update algorithms in the Synthesizer Generator. Work really needs to be done to make the proofs-as-programs approach more efficient.

This section has compared attribute grammars and proofs-as-programs as frameworks

for the definition of the ML editor, *CYNTHIA*. Advantages of the proof approach have been identified, tempered by a number of caveats.

9.2 CENTAUR

The previous section described aspects of the Synthesizer Generator for generating language-based editors. Another approach to generating such editors is embodied in the CENTAUR [Borras *et al.* 88] system. CENTAUR is a system which produces an interactive environment given a specification of the language's concrete and abstract syntax and (optionally) its semantics. Unlike attribute grammars, the various components that define a language need to be expressed in different formalisms — the syntax is written in METAL, the pretty-printer is written in PPML, the semantics is expressed in a formalism based on Natural Deduction called Natural Semantics [Kahn 87] and the graphics interface is written using yet another package (ESTEREL). On the other hand, CENTAUR creates very modular editors, one of the main aims being to connect independent units such as a pretty-printer and a type checker.

By giving a definition of the syntax and pretty-printer, CENTAUR can generate a structure editor in much the same way as attribute grammars. The main difference is the way in which they deal with semantics. CENTAUR users write TYPOL programs where TYPOL is an implementation of Natural Semantics. A TYPOL program is an unordered collection of axioms and rules of inference which can encode information about the static semantics of the language (e.g. to define a type-checker) or the dynamic semantics (e.g. to define an interpreter). The following is an example of a dynamic semantics rule in Natural Semantics for an assignment $ID := EXP$.

$$\frac{s \vdash ID \Rightarrow x \quad s \vdash EXP \Rightarrow v \quad s \vdash^{update} x, v \Rightarrow s_1}{s \vdash ID := EXP \Rightarrow s_1}$$

Once written, TYPOL specifications are compiled into Prolog code (Horn clauses). This way of defining semantics seems flexible and powerful. Obvious applications are for producing an interactive editor which acts as a test-bed for a new semantics definition for a language (as is described for the Sisal language [Attali *et al.* 98]) or for a direct application of semantics for languages with a formal semantics definition

(such as ML).

So far, the main use of TYPOL specifications has been to implement an evaluator for a language from a formal semantics definition. *CYNTHIA* does not concern itself with evaluation of functions. TYPOL has not been used to provide termination checking. Most of the arguments that were used to show the advantages of *CYNTHIA* over attribute grammars can also be used when discussing TYPOL. The main advantage that TYPOL has over attribute grammars is that the use of the semantics rules is a more principled way of encoding semantics. On the other hand, the TYPOL rules do not seem to be integrated very well with the rest of the programming environment. The use of proofs-as-programs in *CYNTHIA* brings an integration that enables the results of semantic analysis to be fed back to the user in a natural and flexible way — the obvious example is when a failed application of an inference rule in the proof tree immediately allows *CYNTHIA* to highlight a type inconsistency in the program. Another example, although not yet implemented, would be to provide feedback in the case of non-termination, such as where the termination analysis has broken down. Such a mechanism could also encompass the suggestion of a possible patch. I believe that this would be more difficult to do with TYPOL rules because of the lack of integration.

A big problem with CENTAUR is that it is generally inefficient. One of the reasons for this is the use of Prolog to execute the TYPOL specifications. Some attempts were made to replace the Prolog code by an attribute grammar evaluator [Attali & Parigot 94]. This would allow the sophisticated, incremental evaluation techniques developed for attribute grammars to be used to improve the performance of CENTAUR.

A number of other editor generators have been developed. Some of these are PSG [Bahlke & Snelting 92], the ASF+SDF Meta-environment [Klint 93], and the ALOE generator of the Gandalf project [Habermann & Notkin 86]. These generally provide a functionality similar to CENTAUR.

9.3 Related Systems

This section describes some state-of-the-art editors, one for Prolog and two for ML. A comparison with *CYNTHIA* is made.

9.3.1 The Recursion Editor

The recursion editor (RE) [Bundy *et al.* 91] was devised by Alan Bundy and implemented by Gerd Grosse under the supervision of Paul Brna[Grosse 89]. Like *CYNTHIA*, the recursion editor is an editor in which only well-defined and terminating programs can be defined. The user starts with a schematic representation of a program and then applies pre-defined editing commands (given in Appendix F) to transform and instantiate the schema. RE is intended for writing Prolog programs but the closed world assumption in Prolog makes it problematic to check for well-definedness so RE uses a declarative style similar to Prolog but excluding this assumption. Definitions are translated to Prolog once complete. Whereas *CYNTHIA* uses proofs-as-programs, RE has no underlying framework for guaranteeing correctness but relies on the fact that each editing command is defined by a (set of) schematic rewrite rule(s) each of which preserves well-definedness and termination.

Example

Suppose the user wishes to write a predicate version of append and has already renamed the initial schema to[1]:

```
append([], L) = ν(L)
append([H|T], L) = ξ(H,T,L,append(T,L))
```

where Greek letters denote as yet uninstantiated function symbols and upper-case Roman letters denote Prolog variables. In Prolog, append needs 3 arguments, 2 of which should be recursive. The user may invoke the command "Add recursion argument" to introduce a second recursive argument giving:

```
append([], L, β) = ν_b(L)
append([], L, γ(Ψ,Θ)) = ν_s(L,Ψ,Θ,append([],L,Θ))
append([H|T], L, β) = ξ_b(H,T,L,append(T,L,β))
```

[1] The code given here is not Prolog code but is RE's representation which is in a functional style. Once the program is complete, RE translates this representation into Prolog.

$$\texttt{append([H|T], L, } \gamma(\Psi,\Theta)) = \xi_s(\texttt{H,T,L,}\Psi,\Theta,\texttt{append(T,L,}\gamma(\Psi,\Theta)),\ \texttt{append([H|T],L,}\Theta),\texttt{append(T,L,}\Theta))$$

β is a new base constructor. γ is a new step constructor with recursive parameter Θ and non-recursive parameter Ψ. Note how new recursive calls have been introduced. Recursive calls can only be introduced by an editing command — they cannot be typed in directly. RE will give recursive calls that could be used without introducing non-termination, but will exclude other recursive calls that could be present under certain restrictions. The above code now needs to be instantiated further to complete the definition.

The schematic rewrite rule corresponding to "Add recursion argument" is as follows. Each old case:

$$\mathit{Condition} \rightarrow \mu[\sharp\sharp[\theta\theta]] = \xi[\mu[\theta\theta]] \tag{9.1}$$

where *Condition* may be empty and where $\sharp\sharp[\theta\theta]$ is a typical recursion argument of μ in the head and where $\theta\theta$ is a corresponding recursion argument of μ in the body, is replaced by two new cases:

$$\mathit{Condition} \rightarrow \mu[\sharp\sharp[\theta\theta],\beta] = \xi_b[\mu[\theta\theta,\beta]] \tag{9.2}$$

$$\mathit{Condition} \rightarrow \mu[\sharp\sharp[\theta\theta],\sharp'(\Psi,\Theta)] = \xi_s[\Psi,\Theta,\mu[\theta\theta,\sharp'(\Psi,\Theta)],\mu[\sharp\sharp[\theta\theta],\Theta],\mu[\theta\theta,\Theta]] \tag{9.3}$$

for some new constructor functions, β and $\sharp'$, recursion variable, Θ, and constructor parameter, Ψ.

As can be seen from the above example, the requirement that each rewrite preserves correctness, whilst providing the maximum scope for further edits, means that the rewrite rules quickly become very complex. It is a major task for the designer of such a system to write these rules and it is very difficult to be sure that the rules are correct — in the sense that they preserve well-definedness and termination and that all cases have been considered. This makes it very difficult for the system to be extended with new editing operations — each one must have a corresponding schematic rewrite which again threatens the correctness of the system. The proofs-as-programs architecture in *CYNTHIA* overcomes this difficulty. The inference rules used in *CYNTHIA* proofs are defined once and for all. First, the replay of rules when an edit is applied means

that the definition of editing commands is much simpler — the above command would correspond merely to a change of specification and a change of the induction rule, the rest of the proof remaining the same. Second, when adding new commands, the designer need not be as concerned with the correctness aspects. The definition of the command will involve the manipulation of an abstract rule tree, but an incorrect program can never be produced because the unfolded rule tree is applied in the *Oyster* proof checker and will fail if any inconsistencies are present.

The underlying framework of schematic rewrite rules also makes the recursion editor very brittle. The order that editing commands are applied is crucial to success. If edits are applied in the wrong order, it is possible to reach a dead-end, at which point the user must scrap the current definition and start again. There follows some examples of this.

Example

The first example arises because recursion can only be introduced using the recursive commands (see Appendix F). Suppose the user is trying to write a version of `delete` and has got to the following point:

```
delete(X, []) = []
delete(X, [H|T]) = ξ(X,H,T,delete(X,T))
```

If the user now manipulates the second result to represent the case when `X=H`, he will have:

```
delete(X, []) = []
delete(X, [H|T]) = T
```

He can now apply "Multiply Cases" to introduce the condition:

```
delete(X, []) = []
X=H => delete(X, [H|T]) = T
not(X=H) => delete(X, [H|T]) = T
```

Note that there is now no way to re-introduce a recursive call into the third clause of the definition. Recursive calls can only arise from modifying existing recursive calls. There is no command for introducing an arbitrary recursive call since that could potentially introduce non-termination.

To complete the definition, the user needed to apply "Multiply Cases" first to give:

```
delete(X, []) = []
X=H => delete(X, [H|T]) = ξ(X,H,T,delete(X,T))
not(X=H) => delete(X, [H|T]) = ξ(X,H,T,delete(X,T))
```

Then the instantiations for each case could be made without difficulty. These kind of order restrictions do not occur with *CYNTHIA*. The user will never get into a situation where the target definition cannot be reached (although in unusual cases, he might have to go a long way round to get there — see Chapter 10, p. 257). The reason for this is twofold. First, the flexibility that comes with proofs-as-programs and second, the fact that the editing commands are better designed.

The scope of programs that can be written using RE is significantly smaller than the scope of *CYNTHIA*. The main restriction is that certain operations must be outlawed in case they compromise termination.

Example

RE prevents the user from introducing recursions based on previous definitions, such as the `qsort` example, from Chapter 5:

```
fun qsort nil = nil : int list
|   qsort (h::t) = qsort (partition (op <) h t) @ [h]
                        @ qsort (partition (op >=) h t);
```

The reason for this restriction is that RE has no general mechanism for checking termination but instead relies entirely on the structure of the editing commands. In fact, what tends to happen is that to prevent non-terminating definitions, RE must over-compensate. For example, RE does not allow nested recursions such as:

```
f(0, Y) = Y
f(s(0),Y) = Y
f(s(s(X)), Y) = f(s(X),Y) + f(X,f(s(X),Y))
```

This definition is an example of nested recursion where termination still holds. There are plenty of cases of nested recursion where termination fails to hold, however. The RE editing commands cannot distinguish between the two cases. Hence, in order to guarantee termination, RE over-compensates and outlaws a wide variety of perfectly good definitions. *CYNTHIA* has a general termination analysis (Walther Recursion) which allows it to deal flexibly with these kind of functions. Hence, it is able to allow a much wider range of terminating functions.

Restrictions such as these mean that RE is severely limited in the kind of recursive patterns that can be defined. In essence, the only allowable recursions are primitive recursions and those where primitive recursion is slightly generalised to allow the recursion variable to jump by more than a single step.

The other major drawback of the recursion editor is the design of the editing commands. Although a few commands are identical to those in *CYNTHIA*[2], for the most part the commands in RE are too restrictive and inflexible. In fact, the lack of suitable checks can even mean that ill-defined programs can be produced.

Example

The first example concerns the introduction of additional step cases using "Multiply step constructor". This command is intended for defining procedures over datatypes with multiple step constructors. There is a similar command "Multiply base constructor" for base constructors. Using these commands, however, it is easy to create an ill-defined procedure. The reason for this is that the constructors need not be instantiated immediately and once they are instantiated, there is no explicit check for well-definedness. As an example, suppose the user has the following definition:

```
fib(zero) = 1
fib(s(zero)) = 1
fib(s(s(n))) = fib(n) + fib(s(n))
```

The user may apply "Multiply step constructor" to give:

```
fib(zero) = 1
fib(s(zero)) = 1
fib(s(s(n))) = fib(n) + fib(s(n))
fib(♯(n)) = fib(n) + fib(s(n))
```

There is now nothing to stop the user instantiating ♯ to `s` giving an over-defined procedure. *CYNTHIA* gets round this by having the command MAKE PATTERN which is directly linked to the datatype definition of a datatype. Hence, cases are produced for each constructor and so no redundant cases can be added.

Another example of inflexibility in RE's commands is that the user is at no point allowed to enter text freely. To introduce an expression such as `a + b`, the user must introduce a schematic function symbol, using "Insert Function" giving:

[2] The only ones are "Add Parameter" which is the same as ADD ARGUMENT and "Multiply Cases" which is the same as ADD CONSTRUCT(IF THEN ELSE)

```
♯(...,a,b,...)
```

and must then remove any unwanted arguments and instantiate ♯. This is, of course, very inconvenient and time-consuming for the user. Again, though, there is an inherent problem in allowing the free entry of text in RE. The user could type in an expression that would introduce non-termination, and without some general mechanism for assessing whether this is indeed the case, the editor would be powerless to prevent it. *CYNTHIA*, of course, does have such a general mechanism, namely Walther Recursion.

A final point on RE is that RE does allow some forms of mutual recursion, whereas *CYNTHIA* does not. This is because Walther Recursion does not incorporate mutual recursion (although it could be extended).

Summarising, *CYNTHIA* has successfully overcome many of the deficiencies of the recursion editor. Apart from the wider range of terminating programs accepted and the improved analysis techniques, *CYNTHIA* also has a graphical user interface which the recursion editor does not.

9.3.2 MLWorks

MLWorks [Har96] is a commercial interactive development environment for ML. The motivation behind MLWorks is somewhat different to *CYNTHIA*. Users write programs in a traditional text-editor and compile their programs using MLWorks. MLWorks also includes various programming tools such as a tracer, debugger, profiler and a tool for graphically representing expressions. The main interest as far as this thesis is concerned is in the type error feedback that MLWorks gives, for it is slightly superior to that of most compilers such as SML-NJ.

MLWorks uses type inference in the same way as SML-NJ does. There are two main differences in the type error messages, however. First, MLWorks tends to give slightly more information about the types that have been derived so far and about where the inconsistency has arisen. As an example, consider this definition:

```
fun test nil = 0
|  test (h::t) = nil;
```

MLWorks's error message is as follows:

```
1 Type disagreement between match rules.
2 Near: h::t => ...
3    Expected type: 'a list -> int
4    Rule type:   'a list -> 'b list
5     Type clash between int and 'b list
```

In contrast, SML-NJ gives:

```
1 Error: right-hand-side of clause doesn't agree with function
         result type
2  expression: 'Z list
3  result type: int
4 in declaration:
5  test = (fn nil => 0
6           | (h::t) => nil)
```

Note how the fact that MLWorks gives the expected and actual type of the *entire* function (lines 3 and 4) is slightly more helpful. Compare lines 2 and 3 in the SML-NJ output, where the context is not clear. Also, line 5 in the MLWorks output explicitly tells the user where the clash has occurred.

The other thing that MLWorks does is to highlight, if asked, the part of the program text where the error occurred. In the above example, the highlighting would be as follows:

```
fun test nil = 0
|   test (h::t) = nil;
```

MLWorks would highlight [ ] whereas *CYNTHIA* would highlight [ ]. Note that MLWorks's highlighting is not as specific as *CYNTHIA*'s.

In general, the fact that MLWorks uses type inference means that the same confusion caused by type errors seen in SML-NJ users is also going to occur with MLWorks. The highlighting mechanism will not generally pinpoint the exact location of the error. Pinpointing the exact location *is* possible in *CYNTHIA* because *CYNTHIA* has a type declaration to use as a guide. Another (simple) example is if the user typed in:

```
0::[ nil ::nil]
```

Again, there is a stark difference in the level of highlighting given.

9.3.3 CtCaml

CtCaml [Rideau & Théry 97] is the one of the latest applications of the CENTAUR technology described in §9.2. It is an editor for the CamlLight [Leroy 95] implementation of ML. However, CtCaml does not make use of the semantic specification techniques in CENTAUR — rather a structure editor is generated by a syntax specification and semantic tools are provided both by linking to the existing CamlLight compiler and by including completely new components.

CtCaml lies part way between *CYNTHIA* and MLWorks. Unlike *CYNTHIA*, it supports the entire ML language. Unlike MLWorks, editing is a hybrid of structure editing and text entry. In CtCaml, the user may select a language construct from a menu which is then inserted at the current point in the program along with meta-variables or gaps that the user needs to fill in. This can be done either by selecting further items from the menu or by typing directly over the variables. After ESC is pressed, the entered text is parsed. This kind of editing is quite difficult to master. This is in part due to awkward switching between text and structure editing and partly because the menu of ML constructs is cryptic. Because all possible constructs are included, the menu forms a very large tree structure through which it is difficult to navigate.

CtCaml highlights ill-typed expressions in a similar way to MLWorks. Again, the highlighting is not quite as specific as that in *CYNTHIA*. The other big difference is that a definition must be completed and then compiled before type feedback can be given. This contrasts with the situation in *CYNTHIA* where as soon as an ill-typed expression is entered, processing is done and the expression is highlighted.

The other advanced feature that CtCaml possesses is a type explanation facility. The need for type explanation arises because of the convoluted form of reasoning about the inference process that quickly becomes necessary. As an example, consider the following code from [Duggan & Bent 96]:[3]

```
... F y; ... y=(3,x); ... F(z,4.5);
```

Type inference will determine that `x` has type `real`. CtCaml, which uses a type explan-

[3] ; is the ML notation for connecting sequences of expressions, which should be evaluated in left to right order.

ation mechanism similar to that in [Duggan & Bent 96], will provide the explanation for the type of x.

```
(F y) constrains F to be a function with domain type
equated to the type of y.
The equality (y=(3,x)) constrains y to be a product type
whose first component has type int and the second component
type is the same as the type for x.
F(z,4.5) constrains F's domain type to be a product type
whose second component type is real.
Since F's domain type is also equated with y's type, this
transitively constrains x's type to be real.
```

Even on this relatively simple example, it can easily be seen that the kind of reasoning needed to work out why a type was inferred becomes very difficult to keep track of. Explanation systems try to make this easier by providing the reasoning in textual form so the programmer does not have to keep track of it himself. However, explanations can quickly become very difficult to read as can be seen from this example.

The main difficulty is in explaining the unification of polymorphic variables in such a way that the approach scales up without overwhelming the user with too much information. The Hindley-Milner inference algorithm walks over the abstract syntax tree of the program, introducing a new type variable for the type of each program variable, collecting constraints on these type variables and using unification to resolve these constraints. The standard Robinson unification algorithm is generally used [Robinson 79]. The idea of [Duggan & Bent 96] is to modify the inference algorithm to allow explanatory information to be stored as part of type unification. Basically, when type variables are unified, the abstract syntax tree is annotated with the program fragment responsible for the unification.

Figure 9.2 shows the type unifications for the example given earlier. Vertices are labelled by type expressions. A directed edge from a type variable vertex to another vertex represents the instantiation of the variable, labelled by the program fragment which gave rise to the constraint causing the instantiation. Subexpressions are distinguished by a subvertex embedded in the original vertex. The dashed edge represents an indirect instantiation of a type variable. The top line of the diagram gives the initial type variables created to represent the variables in the program fragment.

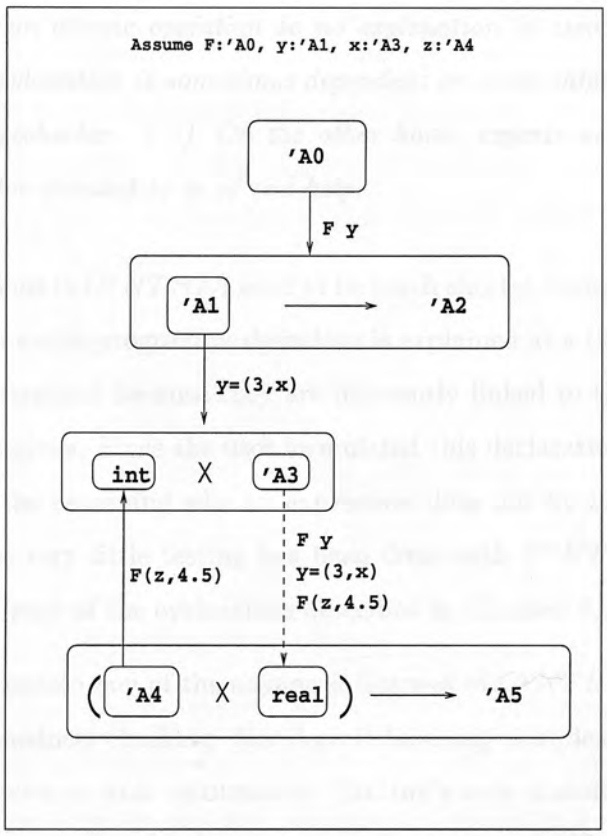


Figure 9.2: Type of F after: `F y; y=(3,x); F(z,4.5).`

Using this graph, the explanation for the type of x is[4]:

```
x : real
F y ............ gives F : 'A0 = 'A1 -> 'A2
=(y,(3,x)) ..... gives y : 'A1 = int * 'A3
F (z,4.5) ...... gives x : 'A3 = real
```

Duggan and Bent also include techniques for making the output more readable, such as eliminating irrelevant type variables. The above output is then translated to the explanation given previously.

Although CtCaml provides an interface for stepping through a type explanation, highlighting the part of the definition corresponding to each sentence in the explanation, the worth of explanation facilities has yet to be shown. Rideau and Théry say:

> *On the one hand it [type explanation] appears to be too complex a tool for being used by a ML newcomer. (...) First, the unification of two types*

[4] where equality has been rewritten as a prefix

> *is treated as an atomic operation so no explanation is associated with it. Second the explanation is sometimes dependent on some internal specificities of the typechecker. (...) On the other hand, experts usually find the explanation too detailed to be of real help.*

The type explanations in *CYNTHIA* tend to be much shorter because only one expression rather than an entire program or definition is explained at a time. They also tend to be easier to understand because they are inherently linked to the type declaration which the user has given. Since the user formulated this declaration, they should find it easier to follow the reasoning why an expression does not fit with it. It has to be said, however, that very little testing has been done with *CYNTHIA*'s explanation facility. It was not part of the evaluations described in Chapter 8, for example.

CtCaml does not contain any of the advanced features of *CYNTHIA* such as termination and well-definedness checking. Nor does it have any complex editing commands such as CHANGE TYPE or ADD ARGUMENT. CtCaml's only transformations are basic ones that replace a meta-variable by a template for a piece of ML syntax.

9.4 Summary

This chapter has examined other frameworks for defining programming environments that can analyse and provide feedback about the semantics of programs. It has also compared *CYNTHIA* to specific programming systems — namely, the recursion editor, MLWorks and CtCaml.

"Well," said Cynthia. "There's your prize." Walter, Cynthia and Mavis were inside the castle. The walls and floor of the castle were a beautiful, sparkling blue colour. Light danced over the knowledges like a will-o'-the-wisp. "It's beautiful," exclaimed Mavis.

"It certainly is," said Walter. "We must be going now. Help yourself to knowledges. And don't get lost on the way back."

"Thanks for everything," said Mavis. She hugged Cynthia and Walter and let them on their way.

Mavis was just about to start gathering knowledges when she noticed a large sign on the far wall. It said: "Maximum five knowledges per person. Additional knowledges may only be taken from the basement." Mavis, like the obediant girl that she was, picked up five largish knowledges and made her way down some steps to the basement. At the bottom of the steps, she could not believe it. The basement was, in actual fact, a huge maze of walls and hidden passages. To find her way to more knowledges she would have to find her way through this maze. "Oh dear," she thought. "It seems like there's a lot more work still to be done!"

Chapter 10

Further Work

This chapter has a look at the future directions in which to take *CYNTHIA*. These include a *CYNTHIA*-like system with stronger specifications, an expert version of *CYNTHIA* and versions of *CYNTHIA* for other languages such as Haskell and Java.

10.1 Immediate Improvements

There are a number of very easy changes that could be made to improve user interaction with *CYNTHIA*. Although I believe the editing commands to be generally well designed, certain transformations can only be done in a sub-optimal manner. For example, suppose the user has written the following code fragment:

```
if e=nil then 0
    else 1 + g e
```

If the user wishes to transform the conditional statement into a `case`-expression:

```
case e of nil => 0
        | (x::xs) => 1 + g e
```

the user has first to apply REMOVE CONSTRUCT to delete the conditional statement and then apply ADD CONSTRUCT to introduce `case`. This is ultimately inadequate. It would be easy to include an editing command that would make this transformation in one step. Another example is if the user has defined a function by pattern-matching on some variable y and wants to move the pattern-matching into the body of the function definition giving something like:

```
fun f x y =
      ⋮
      case y of ...
```

Again, it would be useful to have a command to effect this transformation all at once.

The flexibility of *CYNTHIA* could also be improved even within the subset of ML that is currently supported. It has been mentioned in Chapter 8 that destructor-style definitions are difficult to make in *CYNTHIA*. Although this can be advantageous because it encourages the novice to use constructor-style definitions which are usually the preferred method, there are cases when destructor-style definitions are most natural. *CYNTHIA* should therefore be augmented to deal with destructor style definitions. The mechanisms for allowing this are already in place (for example, Walther Recursion was originally intended by McAllester and Arkoudas to be used with the destructor style anyway) — it is just that the implementation is not mature enough to support it.

The full potential of the editing commands would be realised if *CYNTHIA* was extended to analyse more than one function at once. At present, if, for instance, ADD ARGUMENT is applied, each occurrence of the relevant function *in the current definition* will be modified. Even more useful would be a command that would search through other definitions that depend on the modified function and prompt the user if these should also be updated. This would provide a powerful search-and-replace mechanism based on semantic rather than syntactic considerations. One of the main difficulties in using the proofs-as-programs approach to realise this is efficiency. Applying inference rules in *Oyster* is well-known to be inefficient. So another proof checker would certainly have to be used. It is an open question whether current techniques could provide the sort of performance to achieve such global editing.

10.1.1 Further Evaluation

Chapter 8 described empirical evaluations of *CYNTHIA* carried out at Napier University. Although much interesting and detailed information was obtained from these evaluations, there is clearly a lot more that could be done. Before any further evaluations are undertaken, however, it would be desirable to redesign *CYNTHIA*'s interface

to take into account the students' comments. As stated in Chapter 8, users pay most attention to the usability of the interface rather than to the underlying features. This is because they interact most directly with the interface. It is important therefore to provide an improved interface to make interaction as natural as possible so that further investigation into other features of *CYNTHIA* can be undertaken. The kind of interface that would be useful is outlined in §10.4.4.

There are certain issues that I would like to see investigated more fully. Specifically, these are:

- What kinds of type error feedback is most useful to students? Do students prefer type-checking at compile-time or "on-the-spot" as text is entered? Do students find more advanced features such as type explanation helpful?
- To what extent is a termination checker useful in a programming environment? Is it useful to have an explanation of non-termination in the case that a definition is not accepted?
- What other kinds of explanations for semantic errors could be provided?

Partial answers to these questions have already been provided. I believe the best way to answer these questions would be to conduct more controlled experiments. Then, features that scored highly in these tests could be incorporated into a future version of *CYNTHIA*. The fact that *CYNTHIA* is equipped with many potentially useful features makes it difficult to assess any one of them individually. Certainly, the best way to assess a complete system such as *CYNTHIA* seems to be by informal observations. An alternative to evaluating the system as a whole is to look at specific features. This could enable a more formal analysis to be undertaken. As an example of the kind of experiment I have in mind, consider the question as to whether type explanation facilities are helpful. The idea is that a small "experimental" system be built which would allow a controlled comparison of various kinds of type explanation: no explanation other than compiler error messages; type highlighting; and a full type explanation mechanism such as [Duggan & Bent 96]. A group of subjects, each given only one of these facilities, would be asked to locate the type error in a number of different examples, ranging in difficulty. The time taken to find the source of each

error could be taken along with the time taken to correct it. The experimental system would probably not be based on *CYNTHIA* at all but would be merely an interface to a number of pre-defined examples augmented with the relevant explanation facility. The intention is not to later build on the experimental system but to use it as a vehicle to decide which explanation facility is preferable.

A similar experiment would test what effect forcing the student to give type declarations would have. A version of an ML compiler could be created which insisted that the user provide type declarations. An experiment would then look at the total number of errors made and compare it to those made with a standard compiler and those made with *CYNTHIA*.

10.2 Stronger Specifications

Chapter 5 explained how the strength of the specification in a synthesis proof may vary according to the constraints that are placed on the synthesised object. *CYNTHIA* currently uses a very weak specification describing the input and output types and the existence of reducer/conserver lemmas (if any). Chapter 3 mentioned TED [Bowles *et al.* 94], a system that allows the student to retrieve previous examples based on an English-language specification such as "the output (a list) contains all the elements in the first input (a list)". TED has 16 such specifications which can be used as indices for searching a case base. It is feasible to extend *CYNTHIA* to allow the user to specify stronger specifications that describe relations such as those in TED. Rather than using these relations in searching a database, the idea is that the user would edit programs in the same way as is presently done in *CYNTHIA*, except that the specification of the corresponding proof would be more detailed. Since the specifications are expressed in logic, the approach would be very general. As an example, the relation given earlier, stating that the output list contains all the elements of the first input list, could form a *CYNTHIA* specification:

$$\forall x : int\ list.\ \forall y : int.\ \exists z : int\ list.\ \forall e : int.\ e \in x \rightarrow e \in z \qquad (10.1)$$

Note that it is z that represents the function here. As the user edits a definition, *CYNTHIA* would update and maintain a proof that the definition satisfies this spe-

cification. There are two considerations here. Firstly, it is very unlikely that the program would satisfy the constraints of the specification at each stage of editing. This is the same as happens in *CYNTHIA* currently — intermediate stages of the editing process contain type errors, for example. It is likely that larger parts of the proof would have to be left incomplete until such time as the user provides enough information to complete them. This is the same idea that is used to give feedback on semantic errors in *CYNTHIA*. Secondly, stronger specifications require a more difficult proof to be derived. *CYNTHIA* must be able to prove that any conditions given in the specification hold true. Clearly, we cannot allow arbitrary conditions because the theorem proving task involved could be too difficult or too time-consuming (although one solution to this might be to allow the user to intervene in the proof. This would require a certain level of expertise, however). The specifications would be restricted in such a way that domain-specific tactics could deal with the proof obligations. This kind of tool could certainly be useful for novice programmers. They could write a definition and place a relatively simple constraint on it. *CYNTHIA* would guarantee that their program satisfied this constraint. This would give the novice more confidence in their solution than would have been achieved using only testing.

The specifications in *CYNTHIA* would be less restricted than those in TED. The specifications in TED are all pre-defined by the system designer. However, in *CYNTHIA*, the user could write a much wider range of specifications because of the richness of the underlying logic.

Such a tool could also be useful for expert programmers. Program maintenance is a significant part of the software life cycle. Large programs need to be updated, often by making relatively minor changes. But whilst making these minor changes, the programmer may inadvertently introduce bugs in other parts of the program. If the original programs were given a strong specification of their behaviour then any update of the program could be guaranteed to satisfy this specification as well. Hence, the programmer would automatically be informed of any bugs that he introduced. Because the source and target programs would be very similar, the theorem proving work involved in this task could probably be done automatically. The target proof would be developed by analogy using the source proof as a guide.

10.3 Using a Different Proof Checker

Oyster is a very inefficient proof checker. Although the current speed performance of *CYNTHIA* is usually acceptable, if *CYNTHIA* was used by more advanced students to build up larger programs, a more efficient theorem prover would probably have to be used. This would mean re-implementing *CYNTHIA*'s rules (given in Chapter 6) in a different prover. It is not immediately clear if any other currently available prover would be a better alternative. However, one solution would be to build a theorem prover specially for the purpose. The advantage of doing this is that a logic could be implemented that is specifically geared towards ML. Chapter 6 provides a major step towards this goal anyway. New rules were introduced into *Oyster* that directly correspond to ML concepts or constructs. However, the implementation of certain parts of ML was still hampered — for example, the type systems of *Oyster* and ML are not identical. The current version of *CYNTHIA* has shown that the use of proofs-as-programs is worthwhile. It would be worth investing time in fine-tuning the details of an ML specific logic as gains in efficiency would probably be realised. One way to implement the ML logic would be in a generic proof checker such as *Mollusc* [Richards 93] developed at Edinburgh. *Mollusc* allows the user to define the syntax and semantics for a logic and a proof environment is automatically generated from these.

10.4 An Expert Version of *CYNTHIA*

The current version of *CYNTHIA* has been designed very much with novices users in mind. In principle, however, there is no reason why an improved version would not be a highly useful tool for expert ML programmers. The exact role of *CYNTHIA* would probably be somewhat different. The main reason for using *CYNTHIA* would no longer be for writing smallish function definitions but for maintaining larger programs.

At present, *CYNTHIA* supports only a subset of the ML language. An expert version would clearly need to support the entire language. Many of the omissions in *CYNTHIA* (such as record types) could be included fairly easily. Some features of ML might cause more difficulty, however. These features are discussed below.

10.4.1 Modules

CYNTHIA only supports Core ML — that is, ML without the modules system. The modules system allows a large ML program to be structured into a system of smaller parts connected by simple interfaces. Modules are either *structures* or *functors*. A structure is a collection of declarations. Structures are treated as single units and may be grouped into a hierarchy. A functor is a mapping from structures to structures. Functors allow separate compilation and can express generic definitions. Both structures and functors have a *signature* associated with them. A signature contains declarations for each item in a module. Below is a functor with signature for representing tables as binary search trees (it is a simplified version from [Paulson 91, p. 259]).

```
signature TABLE =
   sig
   type key                          (* type of keys *)
   type 'a T                         (* type of tables *)
   exception Lookup                  (* errors in lookup *)
   val empty : 'a T                  (* the empty table *)
   val lookup : 'a T * key -> 'a     (* look in table *)
   ⋮
   end;


functor TableFUN (structure Order: ORDER and Tree: TREE)
        : TABLE =
  struct

  type key = Order.T;
  type 'a T = (key * 'a) Tree.T;
  exception Lookup;

  local open Tree
    in
    val empty = Lf;

    fun lookup (Br ((a,x),t1,t2), b) =
          if       Order.less(b,a) then lookup(t1, b)
          else if  Order.less(a,b) then lookup(t2, b)
          else x
     |  lookup (Lf, b) = raise Lookup;
    end
    ⋮
```

```
end;
```

It is not important to understand all of the details of this code. Given a structure representing an ordered set and a structure representing trees, the functor will generate tables represented as binary trees with a lookup function.

There is no reason, in principle, why *CYNTHIA* could not be extended to support ML modules, although some changes in the proof structure may be needed to capture the hierarchical aspect of modules. Commands that affect the type of a function in a module would have to update the corresponding signature declaration. For example, if ADD ARGUMENT was applied to `lookup`, *CYNTHIA* would automatically modify the type of `lookup` in the `TABLE` signature. It seems that the current *CYNTHIA* commands would be just as useful when editing modules. Indeed, they may well be more useful. Most functions in modules have an associated type declaration in a signature. This is ideal for *CYNTHIA* because each function must be given a type declaration (although see the next section on type inference). CHANGE TYPE could be applied to functions in the usual way, and the signature declaration would be automatically updated. The function definitions and their type declarations could be far apart in the source file and so a mechanism that automatically kept track of the dependencies could be very helpful.

10.4.2 Type Inference

CYNTHIA does not currently support type inference. An expert version of *CYNTHIA* would need to include this. Fortunately, it appears that type inference can be incorporated into the proofs-as-programs framework very naturally. In fact, there could be some real gains achieved by doing so. I sketch how *CYNTHIA* could be extended with type inference by giving an example. The following is a standard definition of a `delete` function[1].

```
''a -> ''a list -> ''a list
fun delete e nil = nil
|   delete e (x::xs) = if e=x then delete e xs
                       else x :: delete e xs;
```

[1] Recall that `''a` is a polymorphic type that may only be instantiated to a type that admits an equality test.

This is represented as a proof in *CYNTHIA* where the specification consists of the type declaration and any reducer / conserver lemmas (see Chapter 6). Consider only the type part of the specification here. All that needs to be done to include type inference is to start with a specification containing meta-variables, `A`, `B` and `C`:

```
⊢ A → B → C
```

The proof goes through as normal except that all occurrences of the WFFTACS tactic for type checking are replaced by rules for type inference. Applications of these rules instantiate the meta-variables. After initial unquantification, the hypotheses and goal are (only relevant hypotheses are given in the example):

```
e: A, l: B ⊢ C
```

where `l` represents the second argument of `delete`. The next rule applied is the IND rule on `l`. This constrains `B` to be a list and so `B` is instantiated to `D list` where `D` is a new meta-variable. The tactic application produces two children in the proof tree:

```
(1) e: A ⊢ C
(2) e: A, x: D, xs: D list ⊢ 'C
```

The first of these is completed by applying WITNESS with instantiation `nil` which instantiates `C` to `E list`. The IF rule with condition `e = x` is then applied to the second goal, instantiating `A` to `''D`, where `''` indicates that `D` is an equality type. There are further subgoals from this rule:

```
(3) e: ''D, x: ''D, xs: ''D list, e=x ⊢ E list
(4) e: ''D, x: ''D, xs: ''D list, e≠x ⊢ E list
```

Two WITNESS tactics complete these branches. `delete e xs` is introduced in (3) and this type checks. `x::delete e xs` is introduced in (4) which forces the instantiation of `E` to `''D`. Collecting all these instantiations together, the specification has now become `''D -> ''D list -> ''D list`. Replacing `D` with an ML type variable gives the correctly inferred type.

Note that the inclusion of type inference necessitates only very minor changes in the editing commands. MAKE PATTERN is usually applied to a variable whose type is known. In this case, the type is unknown, but the user knows whether the pattern matching should be over lists or natural numbers etc. so MAKE PATTERN can be applied

as normal. CHANGE TYPE is still just as useful for changing the pattern matching, for example from lists to trees.

Programming by analogy works just as before. The user could go on to edit `delete` using *CYNTHIA*'s commands. The only problem with this is that during the incremental editing process, some instantiations of meta-variables will have to be undone and alternative instantiations made. As an example, suppose the user invokes CHANGE TERM to replace `x::delete e xs` by `1 + delete e xs`. This introduces a type error into the program. Any meta-variable instantiations that were caused by the presence of `x::delete e xs` must now be undone. In this case, `E` can no longer be instantiated to `''D` and so *CYNTHIA* would display the program as:

```
''D -> ''D list -> E list
fun delete e nil = nil
|   delete e (x::xs) = if e=x then delete e xs
                       else [ 1 + delete e xs ] ;
```

The type error is highlighted as normal but note that the type declaration is now less specific. In effect, the algorithm has deduced as much type information as possible and relayed this to the user. In general, *CYNTHIA* would have to identify which instantiations need to be undone and which parts of the program need to be type checked again.

This idea seems quite a nice way to provide useful feedback about type errors. The key difference between this mechanism and normal type inference is that rather than just failing if a type error is found and perhaps showing where the conflict occurred, in addition *CYNTHIA* provides a partial type declaration using the information available. Hence, type declarations can be produced for incomplete programs which is not possible with current ML compilers. Some work has been done on trying to extend compilers to produce a type for incomplete programs [Gomard 90] but the potential has not been realised. Note, however, that there would be substantial complications when including the full ML language.

10.4.3 Termination

Another technical difficulty in extending *CYNTHIA* to the full language would be in considering termination. Walther Recursion restricts the user to a subset of ML

programs. Because of the undecidability of termination, no technique is going to be able to provide complete freedom to the user whilst checking termination. However, the role of termination could be changed slightly. The user could have the option of turning off the termination check. In this way, termination is viewed as just another of the programmer's tools that he can choose to invoke as and when he wishes. Alternatively, when the user tries to write a program that cannot be proved terminating, he could assert the program as terminating, given that he was confident of this fact, and then proving termination later on could be done relative to this program.

There is even scope for improving the current termination checker in *CYNTHIA* either by extending Walther Recursion or by combining Walther Recursion with other termination checkers. To illustrate where Walther Recursion could be improved, consider the following examples. The first example is where the fixed size measure used by Walther Recursion is insufficient.

```
'a list list -> 'a list list
fun transp nil = nil
|   transp (nil::xs) = nil
|   transp ((y::ys)::xs) = (map hd ((y::ys)::xs))
                              :: transp (map tl ((y::ys)::xs));
```

`transp` takes a list of lists and returns new lists formed from the first, second, third etc. elements of the lists in the input. For example, `transp [[1,2,3],[4,5,6]]` evaluates to `[[1,4],[2,5],[3,6]]`. `hd` and `tl` return the head and tail of a polymorphic list respectively. `map` is a higher order function that takes two arguments. The first is a function, f, and the second is the list, l. The result of applying `map` to f and l is a list formed by applying f to each element of l. There are a number of points to make about this definition. First, because of the way patterns are constructed in *CYNTHIA* (using MAKE PATTERN) the expression `(y::ys)::xs` appears three times. This is inefficient since the pattern matching could be avoided by replacing `(y::ys)::xs` with a new variable. However, this would lead to the pattern definition:

```
nil
nil::xs
newvar
```

which is not well-defined — the second and third clauses overlap. This pattern set would not be a problem in ML because ML assumes a top-bottom ordering on patterns.

One possible solution would be to replace the third clause in the definition of `transp` with:

```
transp (newvar as ((y::ys)::xs)) = (map hd newvar) ::
                                   transp (map tl newvar);
```

'as' is a piece of ML syntax that declares `newvar` to be a local variable bound to the whole of its argument. This version is not quite as efficient as having no pattern matching but is more efficient than the first version and is perfectly well-defined.

A more serious problem is that `transp` is not Walther Recursive. This example shows the inadequacy of the fixed size measure that Walther Recursion uses. This measure says that the argument of the recursive call must be a list with length strictly less than the length of `(y::ys)::xs`. This is not true, however, because `map tl ((y::ys)::xs)` always has the same length as `(y::ys)::xs`. The definition is terminating, though, because each of the lists in `map tl ((y::ys)::xs)` has length one less than the lists in `(y::ys)::xs`. It would be relatively easy to identify other measures such as this which are still based on the structure of expressions, and augment Walther Recursion to include them. Of course, there is a limit to how far this process can be taken.

Although Walther Recursion can handle many higher-order functions easily, functions that involve partial evaluation in a recursive call are problematic. To illustrate, the usual definition for `foldright` could easily be defined in *CYNTHIA*, because the recursion is just primitive[2]:

```
('a * 'b -> 'b) -> 'a list * 'b -> 'b
fun foldright f (nil, e) = e
| foldright f (x::xs, e) = f (x, foldright f (xs, e));
```

However, the following (somewhat contrived) function is not Walther Recursive:

```
int -> nat -> int
fun f a zero = 0
| f a (s n) = hd (map (f a) (getlist a n));
```

`getlist` is some function with type `int -> nat -> nat list`. The existence of `map` means that there are essentially multiple recursive calls — one for each element of `getlist a n`. Each of these must be measure-decreasing, but the measures of the elements of `getlist a n` are unknown and depend on the definition of `getlist`. Walther

[2] This assumes, of course, that `f` terminates, but since *CYNTHIA* only allows terminating definitions this must be the case.

Recursion could be extended to use lemmas that had been derived for `getlist`, however. Suppose `e` is an element of `getlist a n`. Then we need to derive lemmas of the form:

$$\texttt{e} \in \texttt{getlist a n} \rightarrow \texttt{e} \leq_w x$$

where x is some variable. If `x=n`, then all recursive calls have been shown to be measure decreasing. This is because all recursive calls will have a second argument of size at most that of `n` which is strictly less than the size of `s n`.

As already mentioned, Walther Recursion does not support mutual recursion. It is unclear if Walther Recursion could be extended to deal with mutual recursion. Recursive Path Orderings (RPOs) (see Chapter 3, p. 61) can deal with mutual recursion, however. Ultimately, *CYNTHIA* could be provided with a complementary collection of termination checkers, including Walther Recursion and RPOs, which would extend *CYNTHIA*'s capability considerably. Indeed, in the case that *CYNTHIA* could not show termination, the user could have the option to assert a definition as terminating (assuming he was sure of this) and then any future analogous definition could be shown terminating as well.

Some features of ML mean that it is difficult to show termination. One particular feature that was restricted in *CYNTHIA* is the use of exceptions. An exception may be raised if there is no natural return value (e.g. the tail of an empty list). It is also possible to use the ML keyword '`handle`' to catch the exception and return an actual value. The following is an example of its use:

```
exception whoops;

fun g zero = raise whoops
|   g (s n) = n;

fun h n = g n handle whoops => s n;

fun f zero = zero
|   f (s n) = f (h n);
```

When evaluating `h zero`, an exception is trapped by `handle` and the value `s zero` is returned. When exceptions are used in this way, the normal flow of execution is interrupted and resumes elsewhere. This makes it very difficult to show termination.

The above example does not terminate if `f (s zero)` is called. The exception caught by `handle` could have come from a point far removed. Termination in the presence of exception handlers needs to be investigated.

ML is an *impure* functional language — that is, it has some imperative features, mainly included for input/output. Imperative features make it awkward to analyse a program automatically and showing termination becomes a much more difficult issue.

References in ML are essentially store addresses, corresponding to variables in, for example, PASCAL. They are included for reasons of efficiency. The constructor `ref` creates references. When applied to a value, v, it allocates a new address with v for its initial contents. The function `!` returns the contents of a reference and `:=` is used to assign a new value to the contents of a reference. ML also has imperative control structures, such as `while` for iteration. The following example defines the factorial function iteratively (taken from [Paulson 91]).

```
int -> int
fun ifact n =
  let val resultp = ref 1
      and ip      = ref 0
  in  while !ip < n do   (ip      := !ip + 1;
                          resultp := !resultp * !ip);
      !resultp
  end;
```

Clearly, Walther Recursion cannot be used to show termination of `ifact` as there is no recursion. Techniques do exist for analysing termination of imperative programs but they are unsuitable for an editor like *CYNTHIA* as, in general, the theorem proving effort required is too high.

The imperative features of ML are meant only to be used when strictly necessary. As [Paulson 91] says:

> *These features are not inherently incompatible with functional programming. If they are used in a disciplined manner, large parts of the program will be purely functional.*

Hence, *CYNTHIA* would still find a use in showing termination of the purely functional parts of a program.

Another issue is that of allowing mutual recursions. Walther Recursion does not currently allow mutual recursion. There are two options to including this. First, to extend Walther Recursion. This should be possible. Second, to augment *CYNTHIA* with another termination checker, which would deal with mutual recursion (e.g. use recursive path orders).

10.4.4 The Interface

One of the main obstacles to the use of the current version of *CYNTHIA* is its interface. Editors based on structure editing soon become awkward to use when larger programs are represented. To enable experts to use *CYNTHIA*, the interface should be primarily text based. The user should still be able to call up a menu of editing commands, however. Once selected, an editing command would act in exactly the same way. The main difference is that the user would be provided with the option of using editing commands rather than being forced to use them. To design an editor like this would be a major but interesting piece of research. The only other work that has really come close to this is PSG [Bahlke & Snelting 92].

There are a number of choices as to how such an editor could be implemented. One is to parse the user's text each time an editing command is invoked. This would create a proof representation of the program which could be manipulated and then translated back into text. The other option would be to maintain two versions of each definition — one in the form of a proof and one as plain text. This would eliminate the need for constant parsing of large code fragments but would involve a greater storage overhead.

Type-checking in ML would make the task even more difficult. Type-checking is a relatively time-expensive task. Constant type-checking as the user writes a program could well be too inefficient. The current version of *CYNTHIA* gets around this in two ways. First, only a single definition is checked at once. Second, the analogy mechanism does not replay all type-checking at each edit but only type-checks when strictly necessary. The main difficulty when dealing with large programs is that it would not be sufficient to type-check single functions at a time. The user would wish perhaps to make a change to a definition and then edit another function in a completely different part of the text file. The editor would have to keep track of which functions

had been changed and which other functions were effected so that type-checking could be re-evaluated.

The other difference with this kind of expert editor is that it would make sense to provide the user with a much greater number of editing commands. When working with novices, the over-riding intention was to keep the command set small and simple. With experts, however, commands could be provided for a wide range of different functionalities. By using the system, the expert would gradually decide which of these he found most useful personally and could forget about the others.

10.5 *CYNTHIA* for Other Languages

A natural next step in the development of *CYNTHIA* is to create versions of *CYNTHIA* for other languages to see if the proofs-as-programs idea can be transferred. The most obvious other language to do this for is Haskell [Hudak *et al.* 96] which is another functional language with some key differences from ML. Another possibility is to consider if the approach could be used for imperative languages. Alex Blewitt [Blewitt 98], as part of his MSc work, has written a prototype version of *CYNTHIA* for Java (called JCyn). Key features of *CYNTHIA*, principally termination, have been omitted from JCyn, but Blewitt will continue development of JCyn in his PhD.

The rest of this section describes the essential differences between Haskell and Standard ML and discusses whether any of these are crucial in constructing a *CYNTHIA*-for-Haskell.

ML is an "almost-functional" language, including imperative language features, for instance, to deal with infinite data structures. Haskell is purely functional. Because of its lazy evaluation strategy, non-strict constructors can be used to define infinite data structures such as an infinite list of ones:[3]

```
ones = 1 : ones
```

Clearly, this affects the termination analysis of Haskell programs. One of the criticisms of insisting on termination is that even in ML non-terminating programs are sometimes

[3] Perversely, Haskell uses a different syntax to that of ML. `X : L` means `X` *cons*ed onto `L`. `X::L` means `X` has type `L`.

useful. The answer is to allow the termination checker to be switched off on the understanding that non-terminating programs will only crop up very occasionally. In Haskell, however, non-terminating computations are used frequently. For example, we can compute the Fibonacci sequence efficiently as the following infinite sequence:

```
f a b = a : f b (a+b)
fib = f 0 1
```

Of course, in practical terms, some finite portion of the list is extracted for actual computation. But how is the naive programmer protected from entering code such as[4]:

```
length fib
```

which would not terminate?

As an aside, note that laziness can be simulated in ML by defining

```
datatype 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

where the second argument of `Cons` computes the next element in the sequence or infinite list (see [Paulson 91]). Hence, an infinite list of ones is defined by:

```
fun ones () = Cons (1,ones);
```

Note that ML's type discipline now prevents us from applying `length` to a sequence of ones. One could, of course, still define a length function over `'a seq` that forces evaluation:

```
fun len Nil = 0
|   len (Cons(_,f)) = 1 + len(f());
```

such that `len (ones ())` fails to terminate, but this function has to be specially defined for the purpose. There is no danger of applying `length` to an infinite list. In this sense, there is a clean distinction between finite and infinite lists.

In his invited paper at FPLE95 [Turner 95], David Turner advocates the design of a *strong* functional programming language — i.e. one in which all computations are guaranteed to terminate. His proposed language makes a distinction between finite data types, defined using `data` and infinite data types defined using `codata`. Functions

[4] `length` is the usual recursive length function.

defined on finite data can be checked to be terminating by standard techniques — for example, Walther Recursion. In addition, the idea of using `data` and `codata` places a restriction — namely, that ordinary recursion cannot be applied to codata and corecursion cannot be applied to finite data. By making this separation between finite and infinite data structures, one could never attempt `length fib` because `length` is defined over `data` and `fib` is defined over `codata`.

Haskell does not have this nice separation of finite and infinite data, but it may be possible to somehow keep track of the infinite data within a Haskell program and check whether a function defined over finite data is illegally applied to infinite data. This yields a termination analysis for Haskell based on Turner's ideas.

The object $\perp$ or `bot` in Haskell denotes the value of a non-terminating expression or runtime error[5]. In ML, pattern matching either succeeds or fails. The presence of $\perp$ in Haskell means that pattern matching could also *diverge*. Divergence occurs when a value needed by the pattern contains $\perp$. In practical terms, the ability to define `bot` means that the order of patterns can be crucial. Consider the following example (taken from [Hudak *et al.* 96]):

```
take 0 _       = []
take _ []      = []
take n (x:xs) = x : take (n-1) xs

take1 _ []     = []
take1 0 _      = []
take1 n (x:xs) = x : take1 (n-1) xs
```

`take 0 bot` matches against the first clause and so `[]` is returned. In `take1 0 bot`, however, the second parameter to `take1` needs to be evaluated to see if it is equal to `[]`. This means evaluating `bot` which results in divergence in pattern matching. Hence, the order of the clauses matter. *CYNTHIA* ensures that patterns are well-defined so that the order of the patterns does not matter. In this example, however, there is no way to ensure this because it arises due to the lazy evaluation strategy. A *CYNTHIA*-for-Haskell could still check for well-definedness but would also have to allow re-ordering of clauses according to which result the user required.

A slightly more serious issue is that of *lazy patterns* in Haskell. Lazy patterns delay

[5] For example, we could define `bot` as `1/0`.

pattern matching and are preceded by the symbol ~. Lazy patterns always succeed (in Haskell-speak, they are *irrefutable*). Hence, `[]` will match `~[x]` but the result will depend on the RHS of the equality. Consider the code:

```
f ~[x] = x
```

`f [1]` evaluates to `1` as normal. But `f []` gives an error. This is because `[]` matches the lazy pattern (since anything matches a lazy pattern). However, because of the presence of `x` on the RHS, `[]` must be evaluated. `x` cannot be assigned any value and so there is an error. Contrast the following code:

```
f ~[x] = 0
```

`f []` would return `0`. In this case, `[]` is never evaluated. `[]` matches the lazy pattern as before but there is no need to evaluate `[]`. Instead `0` is just returned.

Lazy patterns can occasionally be useful but are likely to be used infrequently. The immediate consequence of having lazy patterns is that the order of patterns is again crucial. Consider:

```
f [] = ..
f ~[x] = ..
f (h1:h2:t) = ..
```

then `f` evaluates to a different result if the second and third patterns are swapped over. How can this be catered for in *CYNTHIA*? The first question is how can ~ be implemented in the logic? A very pragmatic approach would just define a new rule `ind~` that is the same as IND except that it tags some cases with ~ so the translation from proof to Haskell goes through as planned. Cases where the order of patterns matters can be disallowed by having a new command MAKE LAZY PATTERN. This would transform

```
f :: [Int] -> Int
f x = ..
```

into

```
f :: [Int] -> Int
f [] = ..
f ~(h:t) = ..
```

The order of the patterns is being fixed so that no ambiguities can arise. Note that the lazy patterns will always come at the end of the list of patterns. What happens if MAKE PATTERN is applied to `t`, though? This needs further investigation.

The Haskell type system is very similar to that of ML. The main difference is the provision of *type classes.* Haskell type classes provide a structured way to control overloading (sometimes called ad hoc polymorphism). The basic idea is that some functions are essentially polymorphic in nature but cannot be defined over all polymorphic arguments. Equality is the typical example of this — equality of functions cannot be computed but equality of lists or trees is no problem. In Haskell, one can define an equality class and methods which describe how to compute equality — and this method may differ according to the type. This is achieved as follows in Haskell:

```
class Eq a where
    (==)      :: a -> a -> Bool

instance Eq Int where
   x == y     =  intEq x y
```

There is no problem in including this in a *CYNTHIA*-for-Haskell editor. Clearly, facilities would have to be provided for type declarations and the type checker (or inferer) would have to deal with type classes, but these are implementation tasks. Walther Recursion is not affected. There is a question as to how type classes could be defined in *Oyster* though. The use of subset types seems to be one approach. We could define the above equality class as:

```
{x:u(1)\equal:x→x→bool}
```

the subset of `u(1)` such that there is an `equal` function defined over `x`. Instantiating the class would require a definition of `equal` depending on what `x` is. It seems plausible to define `equal` in the normal way but with one or more casesplits depending on the type of `x`. This might lose slightly the hierarchical nature of the classes, but it does seem possible to express type classes in *Oyster.*

It seems that although there are some key differences between ML and Haskell, none of these are a major obstacle when it comes to producing a *CYNTHIA*-like editor for Haskell. The main change would be in detecting when ordinary recursion is erroneously applied to an infinite data structure.

10.6 Summary

This section has presented some possible future directions for *CYNTHIA*. The most obvious development would be to extend *CYNTHIA* to the full ML language. Other possibilities would be to strengthen the correctness guarantees that *CYNTHIA* provides or to develop *CYNTHIA*-like systems for other languages.

Mavis did eventually find her way into the maze. She took back enough knowledges for the whole of her village and was hailed as a local heroine.

Mavis is now the Chief Village Elder in her village. She has passed on the experience she gained in MikaLand to countless, young hopefuls.

Walter disappeared a few weeks ago. It is believed that he began an attempt to discover what lies at the bottom of the hole in his cottage. Although he has not yet reappeared, visitors to the cottage speak of strange shrieks of "Eureka!" coming from down below.

Cynthia just last week married a woodcutter who lives on the far side of the island. She is already pregnant with their first son. And what will the baby be called? Jason, of course.

Chapter 11

Conclusion

This thesis has described the ML editor *CYNTHIA* which is based on proofs-as-programs and programming by analogy. The contributions of this thesis were set out in Chapter 1. In this final chapter, I examine these contributions in more detail.

11.1 Application of Proofs-as-Programs

The proofs-as-programs idea has been around for some time but it has not previously been used as part of a programming environment. In *CYNTHIA*, proofs-as-programs has been used to represent novice ML programmers' function definitions in a way that the underlying proof framework has been hidden from the user. There have been very few other systems that use formal methods in this way. Systems using formal methods are generally for analysing large programs and hence require a great deal of expertise (in logic and automated theorem proving (ATP)) to be operated. One of the few systems which does hide the underlying framework is the AMPHION system [Lowry & Van Baalen 97] in which users write graphical specifications of physical problems in the domain of space exploration and the system automatically extracts Fortran programs from a proof of these specifications. Although in a different domain than *CYNTHIA*, AMPHION is similar in spirit in that the logic and proof framework is hidden from and need not be understood by the user. The idea in *CYNTHIA* — to exploit proofs-as-programs in an editor for functional programming — is new.

In realising *CYNTHIA*, various contributions have been made. These are:

- Inference rules were developed that directly mirror the programming constructs and concepts in functional programming.
- An analogical mechanism was designed to selectively replay source rules in order to produce a new proof reflecting the user's edits.
- Higlighting of program inconsistencies is enabled by utilising information about where the analogy breaks down. Incomplete proof fragments correspond directly to program errors.
- The technique Walther Recursion was extended to deal with a wide subset of the ML language and rules were devised to implement Walther Recursion in the proofs-as-programs framework.
- An interface was designed that hides the underlying proof information from the naive user. The close correspondence between proofs and programs means that positions in the program can be directly associated with positions in the proof so that from the user's point of view, his edits are apparently applied to the program rather than a hidden representation of the program as a proof.
- An algorithm was developed that can transform pattern definitions according to a change of type between arbitrary types in a subset of the ML types. The correctness of this algorithm was proved.

11.2 A Novel Programming Editor

CYNTHIA stands in its own right as an original and state-of-the-art ML editor, containing various features that are either unavailable in other editors or are improved in *CYNTHIA*. These features are given here.

- Programming by analogy whereby users do not start from scratch but transform an existing program by applying a sequence of editing commands. No other ML editors support programming by analogy although editors for other languages (e.g. Prolog) do exist. However, I believe *CYNTHIA* is superior to these because a much more realistic subset of the target language is supported (i.e. real programs can be written).

- A compact, small set of editing commands was developed. These editing commands are relatively intuitive and easy to learn but do not overly restrict the user to doing things in a particular way.

- Very few other programming environments check termination of programs. Those that do (e.g. the recursion editor [Bundy *et al.* 91]) are severely restricted to a small subset of terminating programs. *CYNTHIA* allows a much wider range of terminating programs to be constructed.

- *CYNTHIA* carries out various other forms of program analysis — type checking, static semantic checking, well-definedness checking. Moreover, the analysis is done incrementally as the program is written rather than waiting until the program is completed.

- *CYNTHIA* provides improved type error feedback over other editors — such as compilers which invariably give cryptic error messages or editors such as MLWorks [Har96] and CtCaml [Rideau & Théry 97] which do have type highlighting but have less specific highlighting than *CYNTHIA* (see Chapter 9 for a discussion).

- *CYNTHIA* provides a structure editor for ML — the only other editor to do this is CtCaml.

There are, of course, some negative comments to make about the interface of the editor. Users sometimes found the editing commands difficult to understand or felt constricted by them. The system's speed of response is sometimes inadequate. The entire ML language is not supported. An editor that was a smooth hybrid of a text and structure editor would restrict the user less. A more efficient proof checker would improve speed. An ML-specific logic would make it easier to support the full language.

11.3 The Psychology of Functional Programming

As well as the two contributions set out so far, work has also been carried out in studying novice programmers. Much of the work in this thesis extends the body of knowledge about the difficulties of learning declarative programming languages. Most

of the existing research has been carried out for logic programming languages such as Prolog. Less has been carried out in functional programming and even less still in ML. The main conclusions drawn from my research in this area are:

- A great deal of suspicion and mistrust exists when novices encounter functional programming. This is because the idea of writing small, individual programs seems strange and far removed their traditional notion of what constitutes programming. Students take a long time to realise and understand the advantages that come with functional programming.

- As with any language, many ML students find it particularly tough-going in the early days. Programming by analogy seems to alleviate the problems of what to put down on to the "blank page".

- As explained in Chapter 2, types are a major source of difficulty for novice functional programmers. Types in ML are substantially different from types in other typed languages. The type inference algorithm can add to the confusion, as can the scarcity of good type error feedback mechanisms.

- Although a well-taught student can understand the simplest pattern definitions, more complicated pattern definitions are often a source of confusion. It is not yet clear if the facilities in *CYNTHIA* for incrementally constructing patterns provide genuine help.

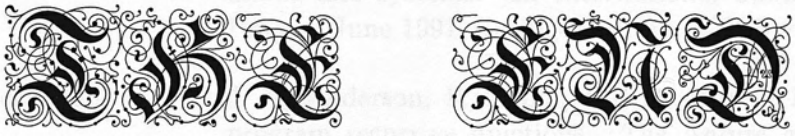
- Students in the early days of programming do not often unintentionally write non-terminating programs. When they do, however, they can be a major obstacle as non-termination is difficult to debug. The extent of usefulness of a termination checker in novice programming is not yet clear. It is possible that students would find termination checking more useful when dealing with more difficult programs containing complex patterns and recursion.

- *CYNTHIA* enforces the accepted ideals of teaching functional programming — starting off with a top-down approach, stressing the structural similarities of programs and using the type of a function as a partial specification. *CYNTHIA* seems an ideal tool to help with the teaching of ML. The editing commands in

CYNTHIA capture the main ideas that need to be learnt and hence provide a common discourse around which a course on ML could be organised. The ease with which *CYNTHIA* was integrated into courses at Napier University suggest that it would be painless to plan courses around *CYNTHIA*.

11.4 Has *CYNTHIA* Achieved its Aims?

To assess whether *CYNTHIA* has achieved its aims, *CYNTHIA* must be analysed on two levels. Firstly, the system successfully hides the proof framework from the user. Trial users of *CYNTHIA* had no idea of the complex analysis going on behind-the-scenes as their edits were reflected in their programs exactly as the users had intended. As a result, users do not require a knowledge of logic or ATP to operate *CYNTHIA*. Moreover, *CYNTHIA* provides a number of features which seem to make it easier for novice users to write programs in ML. Secondly, *CYNTHIA* improves in a variety of ways on existing ML development environments. Students generally make fewer errors when using *CYNTHIA* and those errors that are committed seem to be corrected more easily. The idea of programming by analogy, combined with structure editing, seems helpful, especially to weaker students. In short, *CYNTHIA* is a usable, state-of-the-art editor for ML that manages to seamlessly integrate sophisticated analysis techniques with novel editing facilities.



Bibliography

[Aitken 96] S. Aitken. An analysis of errors in interactive proof attempts. Technical report, Dept. of Computer Science, Glasgow University, June 1996.

[Alblas & Melichar 91] H. Alblas and B. Melichar. Attribute grammars, applications and systems. In *International Summer School*, Prague, June 1991. Springer-Verlag. LNCS v. 545.

[Anderson *et al.* 88] J. R. Anderson, P. Pirolli, and R. Farrel. Learning to program recursive functions. *The Nature of Expertise*, pages 153–183, 1988.

[Attali & Parigot 94] I. Attali and D. Parigot. Integrating Natural Semantics and attribute grammars: the Minotaur system. Research Report 2339, INRIA Sophia Antipolis, 1994.

[Attali 96] I. Attali. Semantic-based visualization for parallel object-oriented programming. *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), ACM Press, Sigplan notices*, 31(10), 1996.

[Attali *et al.* 93] I. Attali, D. Caromel, and M. Oudshoorn. A formal definition of the dynamic semantics of the Eiffel language. In *Proceedings of the sixteenth Australian Computer Science conference (ACSC'93)*, Brisbane, Australia, February 1993.

[Attali *et al.* 98] I. Attali, D. Caromel, and A. Wendelborn. A formal semantics and an interactive environment for SISAL. Research report, INRIA Sophia Antipolis, 1998.

[Bahlke & Snelting 92] R. Bahlke and G. Snelting. Design and structure of a semantics-based programming environment. *International Journal of Man-Machine Studies*, 37(4):467–502, October 1992.

[Barker-Plummer 90] D. Barker-Plummer. Cliché programming in Prolog. In *Proceedings of the META-90 Workshop*, 1990.

[Beaven & Stansifer 93] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, pages 17–30, 1993.

[Bental 95] D. Bental. Why doesn't my program work? : requirements for automated analysis of novices' computer programs. In *Proceedings of the Workshop on Automated Understanding of Novice Programs, World Conference on AI and Education (AIED)*, Washington DC, USA, 1995. Available at http://www.cs.mdx.ac.uk /staffpages /DBental /aiedpaper.html.

[Bhuiyan *et al.* 94] S. Bhuiyan, J. Greer, and G. I. McCalla. Supporting the learning of recursive problem solving. *Interactive Learning Environments*, 4(2):115–139, 1994.

[Bird & Wadler 88] Richard S. Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[Biundo *et al.* 86] S. Biundo, B. Hummel, D. Hutter, and C. Walther. The Karlsruhe induction theorem proving system. In Joerg Siekmann, editor, *8th Conference on Automated Deduction*, pages 672–674. Springer-Verlag, 1986. Springer Lecture Notes in Computer Science No. 230.

[Blewitt 98] A. Blewitt. A Java editor based on proofs-as-programs. Unpublished M.Sc. thesis, Department of Artificial Intelligence, Edinburgh, Scotland, September 1998.

[Borras *et al.* 88] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of SIGSOFT'88, Third Annual Symposium on Software Development Environments*, Boston, USA, 1988.

[Bowles & Brna 93] A. Bowles and P. Brna. Programming plans and techniques. In P. Brna, S. Ohlsson, and H. Pain, editors, *Proceedings of the World Conference on Artificial Intelligence in Education*, pages 378–385. AIED, August 1993.

[Bowles *et al.* 94] A. Bowles, D.S. Robertson, W. Vasconcelos, M. Vargas-Vera, and D. Bental. Applying Prolog Programming Techniques. *International Journal of Human Computer Studies*, pages 329–350, 1994.

[Boyer & Moore 79] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[Brna & Good 96] P. Brna and J. Good. Searching for examples: An evaluation of an intermediate description language for a techniques editor. In P. Vanneste, K. Bertels, B. de Decker,

and J-M. Jaques, editors, *Proceedings of the 8th Annual Workshop of the Psychology of Programming Interest Group*, pages 139–152. 1996.

[Bundy *et al.* 90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[Bundy *et al.* 91] A. Bundy, G. Grosse, and P. Brna. A recursive techniques editor for Prolog. *Instructional Science*, 20:135–172, 1991.

[Bundy *et al.* 93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[Burton 95] C.T.P. Burton. Conceptual structures for recursion. In *Proceedings of Functional Programming Languages in Education*, pages 179–193, Nijmegen, The Netherlands, 1995.

[Card *et al.* 83] S.K. Card, T.P. Moran, and A. Newell. *The Psychology of Human-Computer Interaction.* Erlbaum, Hillsdale, NJ, 1983.

[Casey 94] C. Casey. *A programming approach to formal methods.* McGraw-Hill, 1994. Series in software engineering.

[Clocksin & Mellish 81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer Verlag, 1981.

[Constable *et al.* 86] R. L. Constable, S. F. Allen, H. M. Bromley, *et al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Conway & Gries 79] R. Conway and D. Gries. *An introduction to programming — a structured approach using PL/I and PL/C.* Winthrop, 1979.

[Coquand 92] Th. Coquand. Pattern matching with dependent types. In *Proceedings from the logical framework workshop at Bästad*, June 1992.

[Corbett & Anderson 92] A.T. Corbett and J.R. Anderson. LISP intelligent tutoring system: research in skill acquisition. In J.H. Larkin and R.W. Chabay, editors, *Computer-Assisted instruction and intelligent tutoring systems: shared goals and complementary approaches*, pages 73–109. Lawrence Erlbaum Associates, 1992.

[Damas & Milner 82] L. Damas and R. Milner. Principal type schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, 1982.

[Davison 94] A. Davison. Teaching C in a functional style. Technical Report 94/1, Department of Computer Science, University of Melbourne, February 1994.

[Dershowitz 85] N. Dershowitz. Synthetic programming. *Artificial Intelligence*, 25:323–373, 1985.

[Duggan & Bent 96] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27:37–83, 1996.

[Efremidis *et al.* 93] S.G. Efremidis, K.A. Mughal, and J.H. Reppy. AML: Attribute grammars in ML. Technical Report TR-93-1401, Dept. of Computer Science, Cornell University, 1993.

[Escott & McCalla 88] J. A. Escott and G. I. McCalla. Problem solving by analogy: A source of errors in novice LISP programming. In *Intelligent Tutoring Systems*, pages 312–319, Montreal, Canada, 1988.

[Farmer *et al.* 93] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS : an interactive mathematical proof system. *Journal of Automated Reasoning*, 9(11):213–248, 1993.

[Findler *et al.* 97] R.B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Programming Languages: Implementations, Logics and Programs (PLILP)*, pages 369–388, Southampton, U.K., 1997.

[Foubister *et al.* 98] S. Foubister, G. Michaelson, and N. Tomes. Automatic assessment of elementary Standard ML programs using Ceilidh. *Journal of Computer Assisted Learning*, 1998. To appear.

[Gegg-Harrison 91] T.S. Gegg-Harrison. Learning Prolog in a schema-based environment. *Instructional Science*, 20:173–192, 1991.

[Gegg-Harrison 92a] T. Gegg-Harrison. Adapting instruction to the student's capabilities. *Journal of Artificial Intelligence in Education*, 3:169–181, 1992.

[Gegg-Harrison 92b] T.S. Gegg-Harrison. Adapting instruction to the student's capabilities. *Journal of AI in Education*, 3:169–181, 1992.

[Giesl 97] J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19(1):1–27, 1997.

[GIML] Department of Computing Studies, Napier University, Craiglockhart campus, 219 Colinton Road, EH14 1DJ. *A Gentle Introduction to ML.* http://www.dcs.napier.ac.uk/course-notes/sml/manual.html.

[Gomard 90] C.K. Gomard. Partial type inference for untyped functional programs. In *Proceedings of ACM Conference on LISP and Functional Programming*, pages 282–287, 1990.

[Gordon 88] M. J. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer, 1988.

[Gordon *et al.* 79] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science.* Springer Verlag, 1979.

[Green & Petre 96] T.R.G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.

[Grosse 89] G. Grosse. Towards a recursive techniques editor. Unpublished M.Sc. thesis, Dept of Computer Science, Heriot-Watt University, 1989.

[Haajanen *et al.* 97] J. Haajanen, M. Pesonius, and E. Sutinen. Animation of user algorithm on the web. In *Proceedings of IEEE Symposium on Visual Languages*, pages 356–363, 1997.

[Habermann & Notkin 86] A.N. Habermann and D. Notkin. Gandalf: software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[Hansen 71] W.J. Hansen. Graphic editing of structured text. In R.D. Parslow and R.E. Green, editors, *Advanced Computer Graphics.* Plenum Press, 1971.

[Har96] *MLWorks.* Harlequin, Inc., 1996.

[Hesketh 95] J. Hesketh. Programs as proofs. Tutorial notes, Dept. of Artificial Intelligence, University of Edinburgh, January 1995.

[Hindley & Seldin 86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-calculus.* London Mathematical Society student texts. Cambridge University Press, 1986.

[Holyer 91] I. Holyer. *Functional Programming with Miranda.* Pitman, 1991.

[Homeier & Martin 98] P. Homeier and D. Martin. Mechanical verification of total correctness through diversion verification conditions. In *Proceedings of Theorem Proving in Higher Order Logics (TPHOLs98)*, 1998.

[Howard 80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[Huang 94] X. Huang. Reconstructing proofs at the assertion level. In *Proceedings of the 12th Conference on Automated Deduction*, pages 738–752, Nancy, France, 1994.

[Hudak *et al.* 96] P. Hudak, J. Fasel, and J. Peterson. A gentle introduction to Haskell. Technical Report YALEU/DCS/RR-901, Yale University, May 1996.

[Jun & Michaelson 98] Y. Jun and G. Michaelson. A visualisation of polymorphic type checking. Technical report, Department of Computing and Electrical Engineering, Heriot-Watt University, March 1998.

[Kahn 87] G. Kahn. Natural semantics. In *Proceedings of STACS'87, Lecture Notes in Computer Science*, volume 247. Springer-Verlag, March 1987.

[Kirschenbaum *et al.* 89] M. Kirschenbaum, A. Lakhotia, and L.S. Sterling. Skeletons and techniques for Prolog programming. Technical Report Tr-89-170, Case Western Reserve University, 1989.

[Klint 93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[Kraan 94] I. Kraan. *Proof Planning for Logic Program Synthesis.* Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1994.

[Krathwol 93] D. R. Krathwol. *Methods of educational and social science research.* Longman, New York, 1993.

[Leroy 95] X. Leroy. The CAML Light system (release 0.7). Projet cristal, INRIA Sophia Antipolis, 1995.

[Linn 92] M.C. Linn. How can hypermedia tools help teach programming. *Learning and Instruction*, 2:119–139, 1992.

[Liu 95] Y.A. Liu. CACHET: An interactive, incremental-attribution-based program transformation system for deriving incremental programs. In *Proceedings of 10th In-*

ternational Knowledge Based Software Engineering Conference, pages 19–26, Boston, Massachusetts, November 1995.

[Lowry & Van Baalen 97] M. Lowry and J. Van Baalen. Meta-Amphion: Synthesis of efficient domain-specific program synthesis systems. *Automated Software Engineering*, 4:199–241, 1997.

[Madden 91] P. Madden. *Automated Program Transformation Through Proof Transformation.* Unpublished PhD thesis, University of Edinburgh, 1991.

[Martin-Löf 79] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hanover, August 1979. Published by North Holland, Amsterdam. 1982.

[McAllester & Arkoudas 96] David McAllester and Kostas Arkoudas. Walther recursion. In M.A. McRobbie and J.K. Slaney, editors, *13th International Conference on Automated Deduction (CADE13)*, pages 643–657. Springer Verlag LNAI 1104, July 1996.

[Melis & Whittle 98] E. Melis and J. Whittle. Analogy in inductive theorem proving. *Journal of Automated Reasoning*, 22(2), 1998.

[Michaelson 95] G. Michaelson. *Elementary Standard ML.* UCL Press, November 1995.

[Michaelson 96] G. Michaelson. Automatic assessment of functional program style. In *Australian Software Engineering Conference*, pages 38–46, Melbourne, July 1996. IEEE Computer Society Press.

[Milner *et al.* 90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Neal 89] L. R. Neal. A system for example-based programming. In K. Bice and C. Lewis, editors, *Proceedings of Human Factors in Computing Systems (CHI-89)*, pages 63–68, 1989.

[Ormerod & Ball 96] T.C. Ormerod and L.J. Ball. An empirical evaluation of TEd, a techniques editor for prolog programming. *Empirical Studies of Programmers: Sixth Workshop*, pages 147–161, 1996.

[Owre *et al.* 92] S. Owre, J. M. Rushby, and N. Shankar. PVS : An integrated approach to specification and verification. Forthcoming, SRI International, 1992.

[Owre *et al.* 96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS : Combining specification, proof checking and model checking. In *Proceedings of 8th International Conference on Computer Aided Verification (CAV '96)*, pages 411–414, New Brunswick, NJ, July 1996. Springer Verlag LNCS 1102.

[Paakki 95] J. Paakki. Attribute Grammar paradigms — a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

[Pain & Bundy 87] H. Pain and Alan Bundy. What stories should we tell novice PROLOG programmers. In R. Hawley, editor, *Artificial Intelligence Programming Environments*, pages 119–130. Ellis Horwood, 1987. Also available from Edinburgh as DAI Research Paper 269.

[Paulson 86] L.C. Paulson. Natural deduction as higher order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Paulson 91] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[Pirolli & Anderson 85] P. L. Pirolli and J. R. Anderson. The role of learning from examples in the acquisition of recursive programming. *Canadian Journal of Psychology*, 39:240–272, 1985.

[Pol96] *PolyML.* Abstract, Inc., 1996.

[Protzen 95] M. Protzen. *Lazy Generation of Induction Hypotheses and Patching Faulty Conjectures.* Unpublished PhD thesis, Technische Hochschule Darmstadt, Darmstadt, Germany, February 1995.

[Reps & Teitelbaum 89] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, New York, 1989.

[Richards 93] B. L. Richards. Mollusc user's guide version 1.1. DAI Technical paper 23, University of Edinburgh, September 1993.

[Rideau & Théry 97] L. Rideau and L. Théry. An interactive programming environment for ML. Rapport de Recherche 3139, INRIA Sophia Antipolis, March 1997.

[Rittri 89] M. Rittri. Using types as search keys in function libraries. In *Proceedings of ACM Conference on Functional Programming Languages and Computer Architecture*, London, Sept. 1989.

[Robertson 91] D. Robertson. A simple Prolog techniques editor. Technical Report 523, Department of Artificial Intelligence, University of Edinburgh, 1991.

[Robinson 79] J. A. Robinson. *Logic: Form and function. The mechanization of deductive reasoning.* Edinburgh University Press, 1979.

[Runciman & Toyn 91] C. Runciman and I. Toyn. Retrieving reusable software components by polymorphic type. *Journal of Functional Programming*, 1(2):191–211, April 1991.

[Sack & Soloway 92] W. Sack and E. Soloway. From PROUST to CHIRON: ITS design as iterative engineering. In J.H. Larkin and R.W. Chabay, editors, *Computer-Assisted instruction and intelligent tutoring systems: shared goals and complementary approaches*, pages 239–274. Lawrence Erlbaum Associates, 1992.

[Sannella & Tarlecki 86] D.T. Sannella and A. Tarlecki. Extended ML: an institution-independent framework for formal program development. LFCS Report ECS-LFCS-86-16, Department of Computer Science, University of Edinburgh, 1986.

[Sannella & Tarlecki 91] D.T. Sannella and A. Tarlecki. Extended ML: past, present and future. LFCS Report ECS-LFCS-91-138, Department of Computer Science, University of Edinburgh, 1991.

[Sannella 90] D. Sannella. Formal program development in Extended ML for the working programmer. In *Proc. 3rd BCS/FACS Workshop on Refinement*, Springer Workshops in Computing, pages 99–130, 1990.

[Slind 96a] K. Slind. Derivation and use of induction schemes in higher-order logic. In *Proceedings of Theorem Proving in Higher Order Logics (TPHOLs97), LNCS 1275*, 1996.

[Slind 96b] K. Slind. Function definition in higher order logic. In *Proceedings of Theorem Proving in Higher Order Logics (TPHOLs96), LNCS 1125*, 1996.

[Stallman 79] R.M. Stallman. EMACS: the extensible, customizable, self-documenting display editor. AI Memo 519, Artificial Intelligence Laboratory, MIT, 1979.

[Sutinen *et al.* 97] E. Sutinen, J. Tarhio, S.P. Lahtinen, A.P. Tuovinen, E. Rautama, and V. Meisalo. Eliot – an algorithm animation environment. Technical Report A-1997-4, Department of Computer Science, University of Helsinki, 1997.

[Taylor & duBoulay 87] J. Taylor and B. du Boulay. Learning and using Prolog. Technical Report CSRP-090, School of Cognitive Sciences, University of Sussex, August 1987.

[Taylor 88] J. Taylor. *Programming in Prolog: An in-depth study of problems for beginners learning to Program in Prolog.* Unpublished PhD thesis, School of Cognitive Sciences, University of Sussex, 1988.

[Teitelman & Reps 84] T. Teitelman and T. Reps. The Cornell program synthesizer: a syntax-directed programming environment. In D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 97–116. McGraw-Hill, 1984.

[Teitelman 75] W. Teitelman. *INTERLISP Reference Manual.* XEROX, 1975.

[Thompson 97] S. Thompson. Where do I begin? A problem solving approach in teaching functional programming. In *Proceedings of Programming Languages: Implementations, Logics and Programs (PLILP)*, pages 323–334, Southampton, U.K., 1997.

[Turner 95] D.A. Turner. Elementary strong functional programming. In P.H. Hartel and R. Plasmeijer, editors, *Proceedings of the First International Symposium on Functional Programming Languages in Education, Nijmegen, The Netherlands*, pages 1 – 14, Germany, December 1995. Springer.

[Ullman 94] J. D. Ullman. *Elements of ML programming.* Prentice-Hall International, Englewood Cliffs, N.J., 1994.

[Walther 88a] C. Walther. Argument-bounded algorithms as a basis for automated termination proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 602–621. Springer-Verlag, 1988. Revised version to appear in AI Journal.

[Walther 88b] C. Walther. Automated termination proofs. Technical Report 17/88, Fakultät für Informatik, University of Karlsruhe, 1988.

[Walther 94] C. Walther. On proving termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157, 1994.

[Waters 85] R. C. Waters. KBEmacs: A step toward the programmer's apprentice. Technical Report 753, MIT Artificial Inteligence Laboratory, 1985.

[Weber & Bögelsack 95] G. Weber and A. Bögelsack. *Representation of Programming Episodes in the ELM model.* Ablex Publishing Corporation, Norwood, NJ, 1995.

[Weber 96] G. Weber. Individual selection of examples in an intelligent learning environment. *Journal of AI in Education*, 7(1):3–31, 1996.

Appendix A

A.1 SG1 Questionnaire

1. What type of computing had you done before this course? Mark all that apply as well as your level of skill (before this course) on the scale: 1 = low, 5 = high, 0 = have not used.

- Word Processing
- Spreadsheets
- Statistics
- Programming
- Web Browsers/Internet
- Email

2. Please state any languages you have programmed in before this course and indicate your level of skill on the scale 1=low, 5=high.

3. Recall that programs in CYNTHIA are written by applying a sequence of editing commands to an existing program. How useful did you find it to have an editor that supports this kind of programming technique? (1= Not at all useful, 5=Extremely

useful)

4. How difficult did you find it to transform your starting program to the required program using the editing commands? (1=Very difficult, 5=Very easy)

5. How efficient do you consider CYNTHIA to be as a way of writing ML programs? (1=Not at all efficient, 5=Very efficient)

6. If you had used a traditional text editor rather than CYNTHIA, do you think your progress would have been slower or quicker? Please give reasons for your answer. (1=Much quicker, 5= Much slower)

7. How well do you understand what the editing commands do? (1=Not at all, 5=Completely)

8. Were there any editing commands you found particularly difficult to understand? If yes, please state which and why:

9. Consider the following aspects of CYNTHIA. Please indicate how helpful you found each during program development (1=Not at all helpful, 5=Extremely helpful, 0=Did not come across this):

- **a.** CYNTHIA reduces the amount of text that needs to be typed in
- **b.** Each expression is checked for syntax errors as you enter it, rather than waiting until the program definition is complete
- **c.** The type of a function is constantly visible
- **d.** Clicking the right mouse button on an expression displays the type of the expression
- **e.** Errors are signalled by colouring expressions green or pink
- **f.** Some things are done automatically (e.g. when adding an argument, CYNTHIA adds it to all occurrences of the function)

10. For each of the following error messages, please indicate how easy it was to locate and correct the source of the error (1=Very Difficult, 5=Very Easy, 0=Did not come across this)

- **a.** Syntax error in expression
- **b.** Wrong number of arguments
- **c.** Variable is already in use
- **d.** Type is not valid
- **e.** CHANGE TERM cannot be used to alter recursive calls
- **f.** Program expressions were coloured pink
- **g.** I cannot show termination under this edit
- Other (please state):

11. Did CYNTHIA prevent you making valid changes to your program? If yes, please give details:

12. Are there any additional editing commands you would like to see provided? If so, please give details:

13. Are there any improvements to CYNTHIA you would like to suggest? If yes, please give details:

A.2 SG2 Questionnaire

Answer questions by circling a number in the relevant space and by providing additional information when asked. Please also feel free to add additional comments even if not specifically asked for.

Please state any languages you have programmed in before and indicate your level of skill on the scale 1=low, 5=high :

ML

This section asks you about ML.

Q: How easy did you find ML to learn compared to other courses you have taken?
(1=a lot easier, 5= a lot more difficult) 1 2 3 4 5

Q: Which parts of the course should require more or less attention in the lectures/tutorials?
(1=Much less attention, 5 = Much greater attention)

- the idea of functional programming 1 2 3 4 5
- types in ML 1 2 3 4 5
- recursion 1 2 3 4 5
- pattern matching 1 2 3 4 5
- user-defined types 1 2 3 4 5
- applications of functional programming 1 2 3 4 5
- other (please state):

Tools

This section asks you about tools you used in the course.

Q: How easy did you find it to use the New Jersey SML compiler?
(1=Very difficult, 5=Very easy) 1 2 3 4 5

Q: How easy did you find CYNTHIA to use?
(1=Very difficult, 5 = Very easy) 1 2 3 4 5

Q: Indicate how useful the New Jersey SML compiler was in helping you to:

(1=Not at all helpful, 5 = Extremely helpful)

- identify syntax errors 1 2 3 4 5
- identify type errors 1 2 3 4 5
- identify non-exhaustive or redundant patterns 1 2 3 4 5

Q: What improvements would you suggest to the New Jersey SML compiler?

Q: Indicate how useful CYNTHIA was in helping you to:
(1=Not at all helpful, 5 = Extremely helpful)

- identify syntax errors 1 2 3 4 5
- identify type errors 1 2 3 4 5

Q: How did you like the way of programming in CYNTHIA — of applying editing commands rather than just typing?
(1=Not at all, 5=Very much) 1 2 3 4 5

Q: What improvements would you suggest to CYNTHIA?

Q: Overall, would you prefer to write ML programs using CYNTHIA or a standard text editor? Why?

Please add any other comments you have on the course, ML, or tools in the space below.

A.3 Classification of Errors

A1	Wrong output in a case
A2	Wrong no. args in function definition
A3	Unnecessary no. patterns
A4	Wrong recursive call
A5	Lack of Generality: e.g. condition of `h=3` rather than `h=n`
A6	Too many or too few conditional statements
A7	Wrong condition in conditional statements

Table A.1: Algorithmic Errors

LS1	Duplicate variables in patterns
LS2	Square brackets to make `h::t` a list
LS3	Couldn't put a pair together properly
LS4	Tried to use a variable as a function
LS5	Invalid type
LS6	Misconception about how recursion works
LS7	Misconception about currying
LS8	Gave a function with no argument
LS9	Recursive call has argument missing
LS10	Missed off an argument
LS11	Used `Integer` instead of `int`
LS12	Wrong use of construct

Table A.2: Local Semantic Errors

GS1	Overloaded variable cannot be resolved
GS2	Reference to variables in a different pattern
GS3	Use of a function that's not defined
GS4	Add a variable that's already in use
GS5	Unbound variable or constructor
GS6	Curried function in one place, uncurried elsewhere

Table A.3: Global Semantic Errors

T1	Type Error
T2	Type error in *CYNTHIA* but NJSML accepts a more general version
T3	Wrong top-level type given in *CYNTHIA*

Table A.4: Type Errors

E1	Missed out a case that was necessary
E2	Case missing but a non-necessary case

Table A.5: Pattern Errors

S1	Wrong syntax in use of ML construct
S2	Confusion about outputs: e.g. typing `return 0`

Table A.6: General Syntax Errors

C1	No connection between function calls
C2	Missed out `fun` in definition
C3	Missing bracket
C4	Used a comma instead of a bar for dividing patterns
C5	Incorrect spelling of function / variable name
C6	Used `;;` instead of `::`
C7	No brackets around type declaration
C8	Bad use of comma
C9	Missing whitespace
C10	Missed brackets around pattern

Table A.7: Clerical Errors

U1	Omitted entry in dialog box
U2	Gives pattern, not variable name, to ADD ARGUMENT
U3	Typed entire conditional statement in IF THEN ELSE box
U4	Gave function name for new variable name in ADD ARGUMENT
U5	Used CHANGE TERM instead of IF THEN ELSE
U6	Used CHANGE TYPE to add an argument
U7	Used RENAME instead of CHANGE TERM
U8	Tried to remove an argument that had been split into patterns
U9	Gave top-level type in dialog box for ADD ARGUMENT
U10	Used RENAME instead of MAKE PATTERN
U11	Used ADD ARGUMENT with `nil` instead of MAKE PATTERN
U12	Tried to apply MAKE PATTERN to integers
U13	Tried to add a variable not in use but used internally
U14	Applied CHANGE TYPE to introduce a split of an integer or polymorphic variable
U15	REMOVE CONSTRUCT instead of REMOVE PATTERN
U16	CHANGE TERM instead of ADD RECURSIVE CALL
U17	Use of `case` instead of `if then else`
U18	Only gave arguments, not entire recursive call, as parameter to ADD RECURSIVE CALL
U19	Wrong parameter given to RENAME

Table A.8: Incorrect Usage Errors

J1	Wrong sequence of editing commands because could not get the right command to work
J2	Misunderstood error messages so changed wrong part of program
J3	Error message caused user to incorrectly change part of input in dialog box in *CYNTHIA*
J4	Wrong structure in definition caused by bad source example choice
J5	Misunderstood pink highlighting so changed expression to another ill-typed one

Table A.9: Judgement Errors

A.4 Videoing Experiment Tasks

The students were asked to get through as many of the possible tasks during a 45 minute period. A fourth task was also included but no-one attempted it.

```
1) Write a function that takes a list of zeros and ones and returns
the number of consecutive zeros (if any) at the front of the list.

Examples: leading0s [0,0,0,1,0] = 3

          leading0s [1,0,0,0] = 0


2) Write a function that takes a list of non-negative integers and
returns the maximum integer in the list.

Example:  maxlist [1,3,2,5,3] = 5


3) Write a function which takes two lists of integers and adds
together corresponding elements of the lists.

Examples: addlist [1,2,3] [4,5,6] = [5,7,9]

          addlist [1,2,3] [] = [1,2,3]
```

A.5 Crossover Experiment Tasks

There were three tasks in each part of the test. The tests were carried out under examination conditions. Hints were given to some questions. These were generally intended to give a hint to the correct editing command to use in *CYNTHIA* since it was anticipated that some students would have difficulty choosing commands. However, so as not to bias the results, both *CYNTHIA* and non-*CYNTHIA* users were given the hints. Hence, the hints tend to give a clue as to a construct needed in the solution but give no clue to particularities of the question.

A.5.1 Test X

Please follow the instructions below carefully.

- Write functions to accomplish each of the following tasks.

- Record your answer to each question by typing `answer N;` in ML, where N is the number of the question. E.g. to record your answer to question 1 you would type `answer 1;`.
- If you are using CYNTHIA, the attached sheet reminds you how to save and test functions written using CYNTHIA.
- After 30 minutes, you will be asked to stop.

1) Write a function `addl` which takes 2 arguments: an integer, n, and a list of integers. The function returns a list of pairs. The first component of the pair is the original element. The second component is the element added to n.

e.g. `addl 5 [1,2,3] = [(1,6), (2,7), (3,8)]`

Hint: Note that the arguments are curried. In CYNTHIA, you add an additional argument, by highlighting the name of the function and selecting ADD CURRIED ARGUMENT. Then give the name and type of the argument when prompted.

2) Record your answer to question 1 by typing `answer 1;` in ML.

Write a function `alessnum` which tests element by element whether a list of integers is numerically less than another list. `alessnum` should return `true` if and only if every element in the first list is less than the corresponding element in the second list.

e.g. `alessnum [1,2,3] [2,3,4] = true`

`alessnum [1,2,3] [2,3,2] = false`

Hint: You will need more than two patterns (lines) of code here. In CYNTHIA, you can add an extra line by highlighting a variable and selecting MAKE PATTERN. For example, applying MAKE PATTERN to y in

```
fun f x y = nil;
```

gives:

```
fun f x nil = nil
|   f x (k1::k2) = nil;
```

3) Record your answer to question 2 by typing `answer 2;` in ML.

Write a function that takes a list of integers and produces a new list. The nth element in the new list is the sum of the elements of the old list up to position n.

e.g. `ilist [1,2,3,4] = [1,3,6,10]`

Hint: You may assume the existence of a function `last` to return the last element of a list.

Record your answer to question 3 by typing `answer 3;` in ML.

A.5.2 Test Y

Please follow the instructions below carefully.

- Write functions to accomplish each of the following tasks.
- Record your answer to each question by typing `answer N;` in ML, where N is the number of the question. E.g. to record your answer to question 1 you would type `answer 1;`.
- If you are using CYNTHIA, the attached sheet reminds you how to save and test functions written using CYNTHIA.
- After 30 minutes, you will be asked to stop.

1) Write a function `delete` which deletes all occurrences of an element from a list.

e.g. `delete 3 [1,2,3,3] = [1,2]`

Hint1: Note that the arguments are curried. In CYNTHIA, you add an additional argument, by highlighting the name of the function and selecting ADD CURRIED ARGUMENT. Then give the name and type of the argument when prompted.

Hint2: You may wish to use `if..then..else`. In CYNTHIA, there is a special command to get this. Instead of selecting CHANGE TERM, select ADD CONSTRUCT – IF THEN ELSE. When prompted, type in the condition you want to test. E.g. if you want to write
`if h>x then 0 else 1`, you should input the condition `h>x`.

2) Record your answer to question 1 by typing `answer 1;` in ML.

Write a function combine which takes two lists and returns a new list with corresponding elements combined into a pair.

e.g. `combine [1,2,3] [4,5,6] = [(1,4), (2,5), (3,6)]`

Hint: You will need more than two patterns (lines) of code here. In CYNTHIA, you can add an extra line by highlighting a variable and selecting MAKE PATTERN. For example, applying MAKE PATTERN to y in

```
fun f x y = nil;
```

gives:

```
fun f x nil = nil
|   f x (k1::k2) = nil;
```

3) Record your answer to question 2 by typing `answer 2;` in ML.

Write a function pairlist which takes a list of pairs of integers and returns a list of the alternate components of these pairs.

e.g. `pairlist [(1,2),(3,4),(5,6)] = [1,4,5].`

Hint: You may assume the existence of `fst` and `snd` which return the first and second components of a pair, respectively.

Record your answer to question 3 by typing `answer 3;` in ML.

A.6 Lecturer Report

```
PART ONE
--------

i) What did you personally like/dislike about Cynthia. Any suggested
improvements?

Liked recycling functions. Liked the interface generally.
Disliked mechanism for sending to ML. Disliked type declarations.


ii) How well/badly was Cynthia received by students (please
 distinguish between the groups).
Cynthia was well received by many students and ignored completely
by a minority. My impression was that the post-grads had more
problems - this was due to them being unfamiliar with the solaris
desktop. Most of the undergraduates were (more) used to it.

Also some very early problems with the pg students undermined their
confidence in the system (and mine).

These problems were mostly not Jons.


iii) Which features were most/least helpful to the students?
```

Helpful:
Having a framework to get started with functions.
Unhelpful:
Students had trouble identifying good starting functions.

iv) What aspects of Cynthia helped the students' understanding. Did any aspects confuse them / hold them back?
Cynthia seemed most useful during the early experiences with ML. As they grew in confidence the slightly clumsy interface became more of a hinderance. (By clumsy I simply mean that the mouse needs to be involved at all. Programs get written quicker without them)

v) Did Cynthia in any way affect the way you taught the course?
Cynthia threw up several issues to be discussed in lectures. Types, termination, GUI's.

Mostly I was impressed by how little the teaching material needed to be adjusted. Any more would have constituted dumbing down.

vi) What kinds of students benefitted most/least from Cynthia?
Weaker students benefitted more. It might be more useful to stronger students (and myself) if it were quicker and slicker (keyboard shortcuts esp.)

vii) Did the students generally understand what the editing commands were meant to do? Did they understand the type feedback (i.e. pink highlighting)? Did they find it more or less easy to understand than SML-NJ type error messages? etc...
Nothing could make less sense than the SML-NJ type error messages. Mostly the error messages were helpful. One or two point were the message is wrong. Notably when the type declaration is wrong it insists that there is a type error in the equations.

PART TWO

I have a number of points to make regarding the use of Cynthia in teaching functional programming:

- Recycling functions
- Providing recursive patterns
- Providing an alternative interface to the language
- Assuring termination
- Enforcing type adherance

o Technical limitations
o Critisism of the user interface - details matter

Recycling functions
==========
One of the most interesting features of the system is that it does not permit "new" functions. All functions which are possible in the system can be developed from an existing set of functions.

My belief is that this is entirely appropriate for programmers at all levels. In writting programs we must always be relying on patterns - programming cliches - which we have been taught, or have otherwise appropriated from elsewhere. Very rarely do we create these for ourselves.

Cynthia forces the student to note that the function they require is like another function and to start with that similar function. The same process occurs informally when not using Cynthia. This feature brings into focus a quandary for the teacher: in order to write their own programs the student should be aware of a range of "programming cliches", however the only sure way to fix such cliches in the mind is to use them in writing programs. While Cynthia does not entirely solve this problem it tackles it far more effectively than the traditional system.

The choice of the initial set of functions, and the means by which they can be selected is very important. Ideally the students should be introduced to each of these functions prior to using Cynthia. The teacher should take care to be consistent in the naming of such functions.

As the number of such functions increases it may become necessary to provide a hierachy of "base" functions. Certainly careful thought needs to be given to the process by which the user selects a base function - this is a tangential activity - it is a distraction from the main goal. Such interuptions to the users train of thought should be made as swift and transparent as possible. It might be made possible for the user to view the functions during the selection procedure.

Providing recursive patterns
==========
Cynthia restricts the user to a number of recursive patterns which may be used within a function call. I assume that the motivation is to ensure termination however it clearly has an educational benefit - at least short term.

The third pane lists the recursive patterns which have been analysed and determined to be safe. This may be used by students

however I did not notice anyone taking advantage of this directly. More importantly students are unable to enter incorrect recursive calls. Typically the recursive call chosen initially was suitable and did not need to be changed. If the wrong starting point was chosen then the student was as likely to start from scratch with the right example.

Providing an alternative interface to the language
==
A common concern with students is that SML does not provide the kind of integrated development environment that they are used to with languages such as Borland C++ or Microsoft's Visual Basic.

I have always regarded this as a trivial matter, nevertheless the concern has been reiterated suffiently often to take it seriously. Cynthia provides an appropriate, if relatively primitive integrated development environment.

Assuring Termination
====================
Preventing failure to terminate is undoubtedly the most impressive of Cynthia's features. The value of this in a normal programming environment is clear however there are one or two disadvantages for the learner:

o Lack of negative feedback.
The version of SML in use provides a powerful disincentive to producing non-terminating programs.
Non-terminating programs usually result in impressive looking garbage collection warnings and visible degradation in the performance of the workstaion in question and it's neighbours.
A running gag in my lectures includes the notion that computers which go wrong result in explosions - like they do in Bond films.
A non-terminating function is my cue to start evacuating the laboratory.

o I am undecided as to importance of the limitations that Cynthia imposes on the programmer. Clearly Cynthia is an "insuffiently-powerful" programming language in terms of the Halting Problem. My question is does this matter? For learners following a fixed scheme there is no problem - Cynthia was able to include all of the examples in the notes which the students normally see.

Jon took pains to update Cynthia so that it worked with examples as we went along - I am concerned that this process might not slow down. Would a practical programming tool ever settle down?

Enforcing Type Adherance
========================
There was a dispute over the importance of type for students
learning to program. My feeling is that type is a distraction,
it gets in the way of students producing programs, type should
be the last thing that students worry about.
ML is an attractive language because type declarations are not
usually required.

Jon's feeling was that getting the type right should be one of
the first things considered.

The issue arises because of a common error which users get from
Cynthia. Users frequently have an inconsistency between the type
declaration and the equations given.

My argument is that in many cases it is the type declaration that
is wrong - but Cynthia highlights the equations as being wrong.
It ought to be possible for Cynthia to highlight the type declaration
as being wrong in cases where more than one equation is present and
they are consistent with each other.

That Cynthia requires a type declaration is unfortunate.

Technical Limitations
=====================
There is no doubt that the sluggish performance of Cynthia on our
system was very detrimental. Such programs really need an instant
response. Seeing Cynthia work at a better speed convinces me that
it would have been even more successful had we the hardware.
Our collection of Suns is getting rather elderly.

I was very impressed with Jon's ability to come up with an effective
mono-chrome solution very quickly.

User Interface
==============
Details matter in the user interface - but not all details. When a
text box is brought up it should be filled in with a sensible
default value for the text - but where should the cursor be and
how much text should be selected. Many issues such as these arose
and were discussed and sometimes acted on.

Appendix B

Tutorial and Manual for *CYNTHIA*

B.1 A Quick Introduction to *CYNTHIA*

CYNTHIA is a new programming environment for (a subset of) Standard ML. Programming in *CYNTHIA* is done in a very natural way, so it should not take long to become accustomed to it. *CYNTHIA* is founded on the following two principles.

- Program Transformation. All programs in *CYNTHIA* are created by applying a sequence of editing commands to existing programs. This means that programs can be modified quickly and safely.
- Program Correctness. In *CYNTHIA*, the user is restricted to a subset of ML, namely the subset of functions that are TERMINATING, WELL-DEFINED, WELL-TYPED and SYNTACTICALLY CORRECT. This means that the programmer encounters fewer errors when programming.

B.1.1 What is Termination?

Terminating programs are guaranteed to stop on all possible inputs. Non-terminating programs may go into an infinite loop.

The simplest terminating function would be something like:

```
fun silly x = x;
```

But there are also less obvious non-terminating functions:

```
int -> int
fun half x = if x=0
             then 0
             else 1 + half (x-2);
```

Consider what happens if you type `half 5;`

It has been shown that it is impossible to write an algorithm that can decide for any given program, whether or not it terminates. However, there are useful techniques for deciding termination for a subset of ML programs. *CYNTHIA* uses one of these. The main consequence is that there are some terminating programs that cannot be written in *CYNTHIA*. However, *CYNTHIA* does support a sufficiently wide range of terminating functions for this course.

B.1.2 What are Well-defined Functions?

A well-defined pattern is one in which there are no redundant matches and which is exhaustive. Ill-defined patterns are generally considered bad programming practice and can lead to errors. Therefore, *CYNTHIA* disallows them.

A pattern with redundant matches is given below:

```
fun overdefined nil = ...
|   overdefined (x::xs) = ...
|   overdefined (x1::x2::xs) = ...
```

The third case is redundant since anything that matches it will already have matched the second case.

A non-exhaustive pattern is given below:

```
fun underdefined (h::t) = ...
```

Note that there is not case if the input is `nil`.

B.1.3 What are Well-typed Functions?

Well-typed functions are ones which do not contain any type mismatches or contradictions. ML only accepts well-typed functions. The following is an ill-typed function:

```
fun length nil = nil
|   length (x::xs) = 1+ length xs;
```

The output type in the first pattern is a list, but in the second is `int` (because `+` has type `int -> int -> int`).

B.2 First Time with *CYNTHIA*

This document is intended to be worked through by *CYNTHIA* first-timers. It shows you how to define two simple list-processing functions in *CYNTHIA*. For further information on list recursion, refer to Andrew's tutorial.

B.2.1 Start up

To run *CYNTHIA*, type in the following at a Unix prompt:

```
cynthia
```

B.2.2 Writing `length`

Suppose you want to write a function which counts the elements in a list.

```
length [4,2,5,1] = 4
```

Rather than write this from scratch, use the function, `sum`,which adds all the elements of a list, as a starting point.

```
fun   sum  nil    = 0
|     sum (h::t)  = h + sum t;

sum [4,2,5,1] = 12
```

Doing this in *CYNTHIA*

Click on Select from the EDIT menu. Choose `sum` from the menu. `sum` can be transformed to `length` by making two very simple changes.

Renaming `sum`

First, we should rename `sum` to `length`. Place the mouse over the first occurrence of `sum` and click with the left mouse button. You will be provided with a menu of options. These are the possible edits you can make at this point in the program. Select RENAME from this menu.

A dialog box will appear prompting you to enter a new name for `sum`. Type in `length` and click on OK.

Note that *CYNTHIA* has automatically changed all occurrences of `sum` to `length`.

Counting elements

There is one more change we need to make: can you see what it is?

Rather than adding each element, we just want to count it. It suffices that the output in the second clause should be changed to `1 + length t`

Position the mouse over the `h` in `h + sum t` and click on the left mouse button. Select CHANGE TERM from the menu. A dialog box will appear. In this dialog box, type `1`. The resulting definition should look like this:

```
fun length  nil    = 0
|   length (h::t) = 1 + length t;
```

Trying this out

You can try this out in ML. Start up ML as usual by typing:

```
sml
```

Then do the following:

- Select Save from the FILE menu. This saves your definition of `length` in a format recognised by *CYNTHIA* so that you can load it up and edit at later.
- Select Save as Ascii from the FILE menu. This saves your definition as plain text so you can load it into ML. It also prints the definition into the xterm from which *CYNTHIA* was started. You can now paste this into ML and try out your definition.

B.2.3 Writing `multlist`

Consider the function `doublist` which doubles the elements in a list:

```
fun  doublist  nil    = nil
|    doublist (h::t) = 2*h :: doublist t;

doublist [4,2,5,1] = [8,4,10,2]
```

Consider how to generalise this to `multlist` which takes an additional argument and multiplies each element in the list by this argument.

Doing this in *CYNTHIA*

Select `doublist`.

Rename `doublist` to `multlist`.

We need to add an additional argument. Place the mouse over `(h::t)` and click on the left mouse button. Select ADD ARGUMENT from the menu.

You will be prompted to enter the name and the type of the new argument. Let's call our argument `n` and give it type `int`. Note that to save typing, you can select `int` by clicking on the button to the left of 'Type'.

Multiplying by n

There is one change left to make - we need to multiply `h` by n at each stage of the recursion. See if you can make this change, then save the definition and try it out in ML.

```
multlist ([4,2,5,1], 5) = [20,10,25,5]
```

Now work through Andrew's tutorial on list recursion using *CYNTHIA*.

B.3 Examples

B.3.1 A List Processing Example

Suppose you are given the following task:

Write a function, `addlist` which takes two arguments `x` of type `int list` and `y` of type `int`. The result of applying `addlist` to its two arguments is a new list identical to `x` except that each element of `x` has `y` added to it.

How would you go about writing this function?

A good start is to decide upon the recursion in the function. The idea is to recurse down `x` adding `y` at each stage of the recursion.

Can you think of a similar function that you have seen before?

The recursion in `addlist` is exactly the same as that in `sum` (a function to find the sum of elements in a list). We can use *CYNTHIA* to manipulate `sum` into `addlist`.

Why might we want to do this?

There are two answers really. First, `sum` (assuming it has already been written) provides a starting point so you don't have to write `addlist` from scratch. Second, as you modify `sum`, *CYNTHIA* will check a lot of your edits. Hence, you are likely to make fewer mistakes.

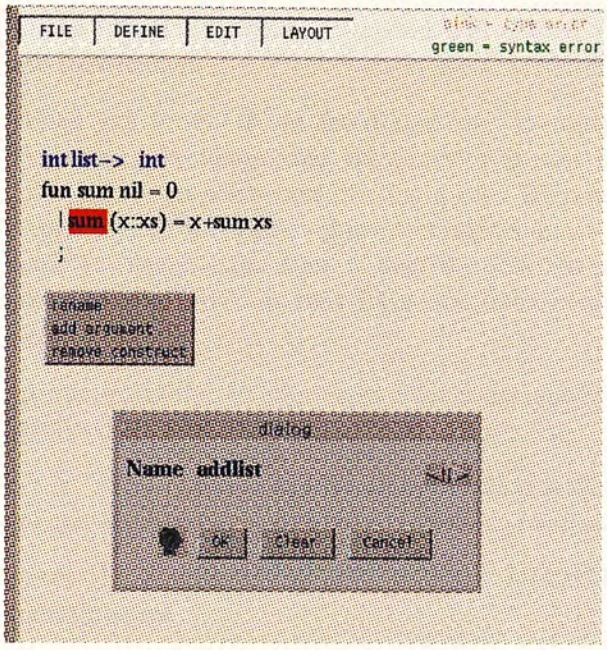
Edit `sum` into `addlist` using *CYNTHIA* as follows:

Go to the EDIT menu and click on Select. This will provide you with a choice of functions that are loaded into *CYNTHIA*. Select `sum`.

CYNTHIA is split into three sections: the EDIT canvas, the MESSAGE canvas and the RECURSION canvas. The EDIT canvas is the white area where the definition of `sum` has appeared. The MESSAGE canvas is the grey area in the middle and is designated for system messages. The green area at the bottom is the RECURSION canvas. It shows all recursive calls that can be used at the present time. For `sum` there is only one possibility at the moment. We will see later how this list can be modified so that more complex recursive functions can be written.

This example illustrates one difference between *CYNTHIA* and the New Jersey SML compiler. In *CYNTHIA*, the type of the function being defined must be given by the programmer. In practice, this is not much of a drawback since types can be modified easily from old definitions. The advantage is that by doing this *CYNTHIA* can provide more help locating type mismatches in your programs.

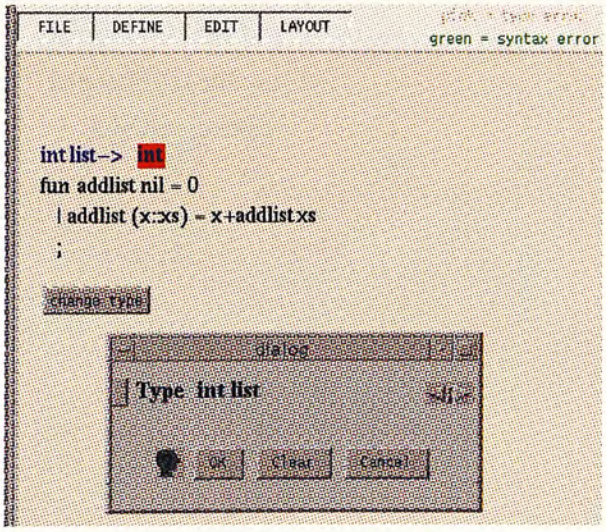
The first modification we should make to `sum` is to change the name. Place the mouse over the highlighted occurrence of `sum` and click on the left mouse button. A menu will appear with possible editing commands. Select RENAME and a dialog box will appear in which you should type the new name, `addlist`.



Note that the new name is propagated throughout the entire program.

What do we need to do next?

Note that `sum` yields an integer as its output. `addlist`, however, will yield an integer list. Therefore, we need to change the type of the function. This can be done easily using the CHANGE TYPE command. Click on the output type of `sum` and again a dialog box will appear:



You can just type `int list` in the dialog box or alternatively, click on the button

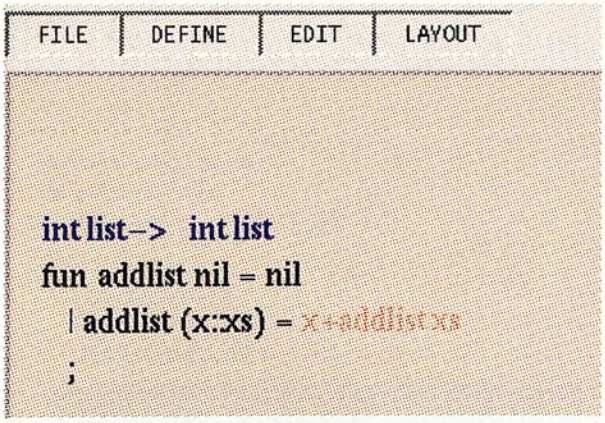
to the left of "Type" and a menu will appear from which expressions can be selected automatically.

When you press OK on the dialog box, the output type will be changed to `int list`. However, this means that the function is now ill-typed - for instance, `0` is not of type `int list`. To warn you of this, *CYNTHIA* will change the colour of the offending expression. You will now have to make changes to the program so that the colour goes back to black. Note that it may be possible (and indeed desirable) to change other parts of the function that have not been coloured to make the function well-typed again.

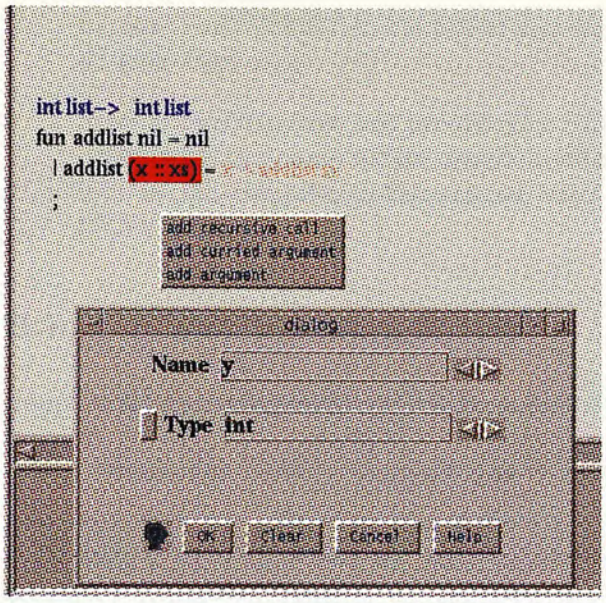
In our example, we can turn `0` black by changing it to `nil`. This is exactly what is required by `addlist`.

CYNTHIA can in fact turn an expression into one of two colours: pink denotes a type mismatch whereas green tells us that the syntax is incorrect.

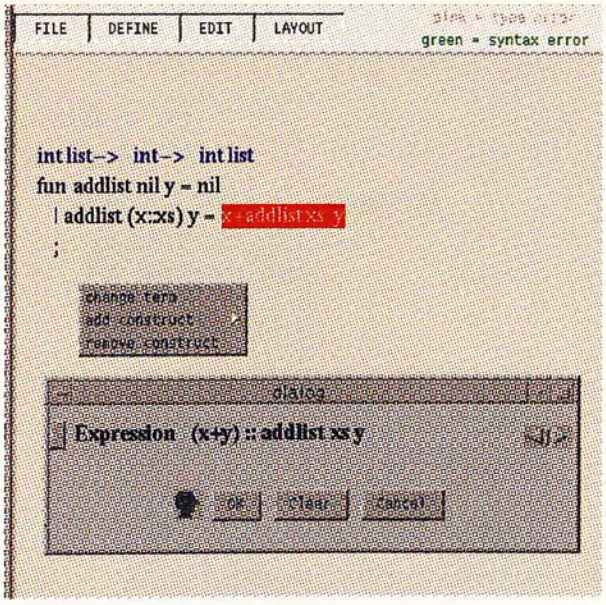
Place the mouse over `0` and press the MIDDLE mouse button. Select `nil` from the menu posted.



We must now change the result in the other pattern. However, before we can do this we need an extra argument, `y`. We can add this automatically - place the mouse over `x::xs` and click. Then select add curried argument from the menu. A dialog box will appear asking you to state the name and type of the new variable.



CYNTHIA will automatically add an argument `y` throughout the function definition. Now all we have to do is to change the result in the second pattern. Click on the old result `x+addlist xs y` and select CHANGE TERM. Now you may edit the old expression in the dialog box in the usual way.



We now have a completed definition for `addlist`.

You can now save the program for use. There are two save options:

- Save: Save the program in *CYNTHIA*'s own format. Such programs can only

be used in *CYNTHIA*. Do this if you want to edit the program in *CYNTHIA* at some later stage.

- Save as ASCII: Save the program in ASCII format to a file. The file will be called with the name of the program with a .sml suffix. You can then load this into an ML compiler in the normal way. Such files cannot be loaded back into *CYNTHIA*.

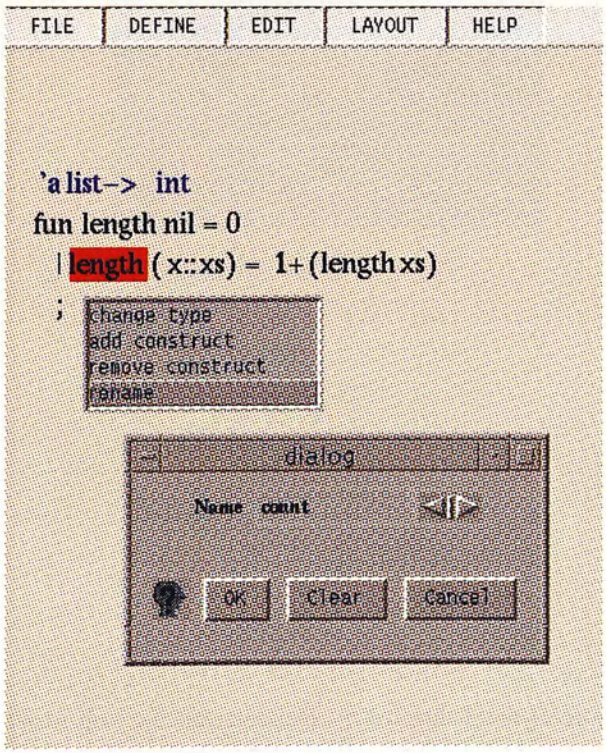
B.3.2 A Tree Processing Example

Suppose the following datatype has been defined:

```
datatype tree = leaf of int | node of tree * tree;
```

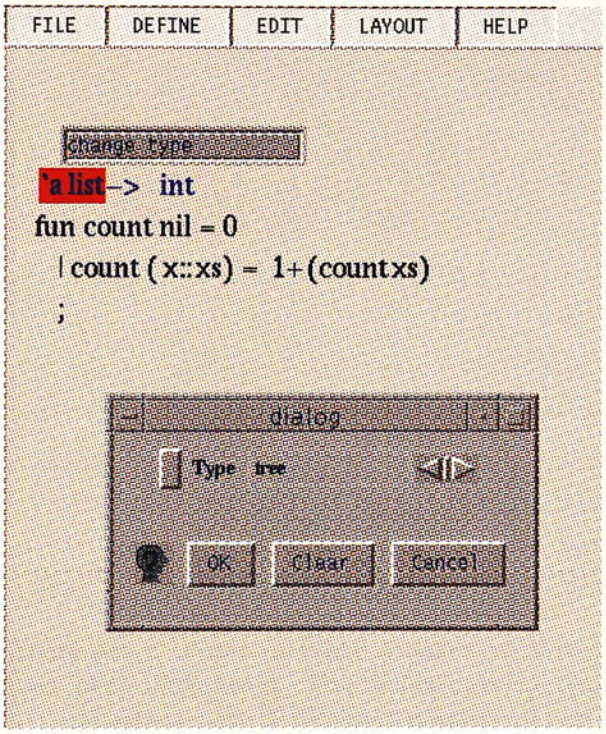
You want to write a function that counts the number of leaf nodes in a given tree. Fortunately, you don't have to write count from scratch in *CYNTHIA*. You have access to a function `length`, for counting the number of elements in a list. These functions are conceptually very similar. This section shows how you can transform `length` into `count` in *CYNTHIA*.

Let's start with the definition of `length`. From the FILE menu, select Open and then load in the definition of `length` that is in the library.



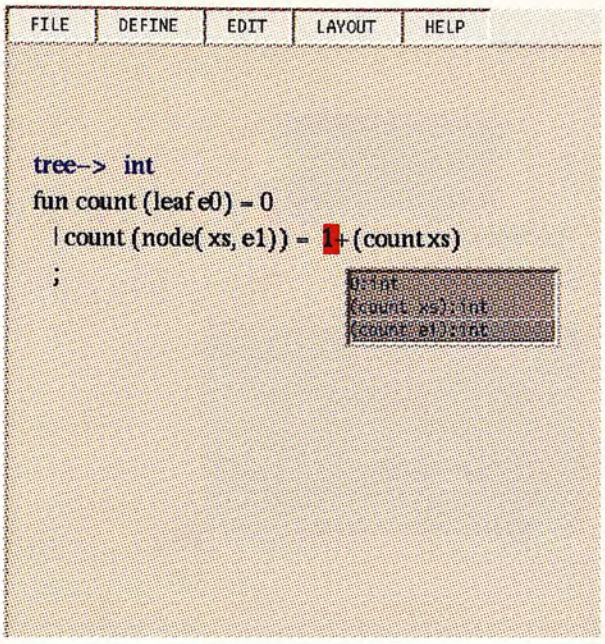
This is the normal definition of `length` which is of polymorphic type and recurses on the list. The first, most obvious thing to do, is to change the name of the function we

are editing. Rename the function by placing the mouse over any occurrence of `length` and clicking on the left mouse button. A dialog box will then pop up in which you can type your preferred name.



Note how all occurrences of `length` have been changed to `count`.

The next thing we need to do is to change the datatype of the function. The input should now be of type tree, not a polymorphic list as before. Place the mouse over `'a list` and click on the left button. Typing in `tree` at the prompt will induce the pattern by which count is defined to be changed throughout the program - note how the constructors `nil` and `::` change to `leaf` and `node`.

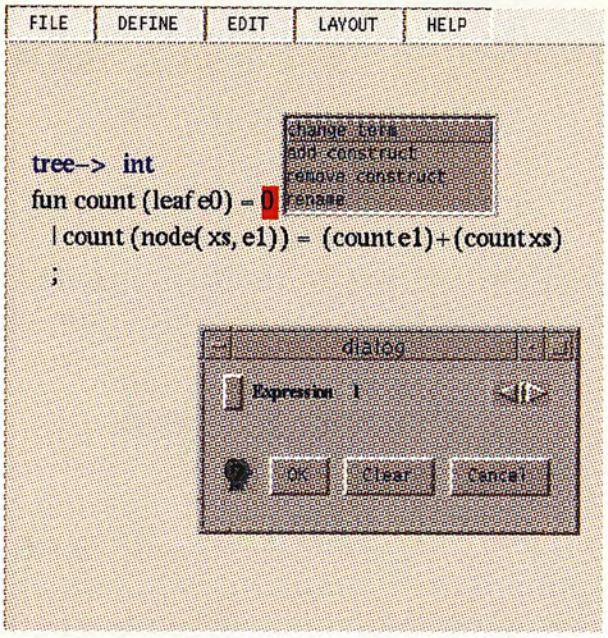


There are a couple of other changes to note. First, two new variables, `e0` and `e1`, have been introduced. This is because the constructors `nil` and `leaf` (and similarly `::` and `node`) have a different number of recursive arguments. *CYNTHIA* will try to match recursive arguments. For `::`, its second argument is recursive. For node, both its arguments are recursive. Hence, rather than retain the variable `x`, *CYNTHIA* introduces `e1` into the program. Had it introduced `x`, we would have had `node(xs,x)` which is not well-typed.

Second, the recursive calls available for use have changed. The bottom part of the display shows which recursive calls are available at any point in time

We are told which recursive calls belong to which pattern. Any recursive calls displayed here can be used in the program. You may add or remove recursive calls from this list as long as no calls are added that could potentially make a function non-terminating.

In this example, *CYNTHIA* has automatically added a recursive call for `(count e1)`. It does this because `e1` is a recursive argument so we may wish (although we don't have to) to use it in some way. In fact, we do need it here. To count the number of nodes in a tree, we count those in the left branch and those in the right branch of all non-terminal nodes. Hence, we need to change the output in the 2nd case. Do this by clicking on the term and editing the expression that is brought up in the dialog box.



All that remains to do is to change the base case to 1, since leaf nodes contain a single node. Do this in the same way as we changed the output in the 2nd case.

We have now successfully defined a count function for binary trees.

B.4 *CYNTHIA* Editing Commands

This section gives a quick description (by example) of each of *CYNTHIA*'s editing commands. Boxed expressions in definitions denote where the command has been applied.

B.4.1 ADD CURRIED ARGUMENT

```
int list -> int

fun sum [ nil ] = 0
|   sum (x::xs) = x + sum xs;
```

Adds a (curried) argument to the current function definition. You will be asked to input the type of the new argument (`int` here). Note how all occurrences of `sum` are given the extra argument.

```
int list -> int -> int

fun sum nil y = 0
|   sum (x::xs) y = x + sum xs y;
```

B.4.2 ADD ARGUMENT

```
int list -> int

fun sum [ nil ] = 0
|   sum (x::xs) = x + sum xs;
```

Adds an (uncurried) argument to the definition. The type of this argument is provided by user. Note that the input argument becomes a pair type.

```
int list * int -> int
fun sum (nil,y) = 0
|   sum (x::xs,y) = x + sum (xs,y);
```

B.4.3 ADD CONSTRUCT

Used to add an ML construct at the current point. *CYNTHIA* supports the constructs `if then else`, `let val`, `let fun`, `case` and `fn`. Each requires user input as follows:

ADD CONSTRUCT (IF THEN ELSE)

```
int list -> int

fun sum x = [ 0 ];
```

The condition `x=nil` is provided by the user.

```
int list -> int

fun sum x = if x=nil
            then 0
            else 0;
```

ADD CONSTRUCT (LET VAL)

```
int list -> int

fun sum nil = 0
|   sum (x::xs) = [ sum xs ];
```

Add a local variable. User is asked to provide:

- Pattern - enter a pattern such as `z` or `(z1,z2)` etc.
- Type - enter the type of the pattern such as `int` or `int * int`

The definition of the local variable is denoted by `???` and can be completed by using other editing commands such as CHANGE TERM.

```
int list -> int

fun sum nil = 0
|   sum (x::xs) = let val z= ???
                       in
                       sum xs
                       end;
```

ADD CONSTRUCT (LET FUN)

```
int list -> int

fun sum nil = 0
|   sum (x::xs) =  [ sum xs ]
```

Similar to `let val` but adds a local function definition. User is asked to provide:

- Name of function - e.g. `double`
- Arguments (variables) of function - `a` here. If there is more than one argument, separate the variable names by spaces.
- Type of the function - `int -> int` here.

```
int list -> int

fun sum nil = 0
|   sum (x::xs) = let fun double a = ???
                       in
                       sum xs
                       end;
```

You can view the type of the local function by double-clicking on the left mouse button over the function name (`double` in this case).

ADD CONSTRUCT (CASE)

```
int list -> int

fun sum x = [ 0 ];
```

Introduces a case construct on the expression provided by the user. Will provide a default pattern for the expression. More complex patterns can be built up using MAKE PATTERN.

```
int list -> int
fun sum x = (case x of
                nil => 0
              | (x::xs) => 0);
```

ADD RECURSIVE CALL

CYNTHIA deals with different recursions in the following way. Suppose we wanted to write an efficient version of a reverse function on lists. The way to do this is to use a second argument as an accumulator:

```
fun rev nil y = y
|  rev (x::xs) y = rev xs (x::y);
```

To reverse `[1,2,3]` we call `rev [1,2,3] nil = rev [2,3] [1] = rev [3] [2,1] = rev nil [3,2,1] = [3,2,1]`. Remember that *CYNTHIA* guarantees termination. Therefore, *CYNTHIA* does not allow you to add arbitrary recursive calls. You have to add recursive calls incrementally using the add recursive call command: Apply this command and when prompted type in the recursive call `rev xs (x::y)`. Note that this term now appears in the lower window in *CYNTHIA*. This window shows the recursive calls that may be used at any one time for each pattern case. We can now add this recursive call into the definition using change term. Note that only terminating recursive calls can be added. Try adding the call `rev (x::xs) (x::y)`. This will cause infinite looping since `rev [1,2,3] [] = rev [1,2,3] [1] = rev [1,2,3] [1,1]` etc.. Therefore, *CYNTHIA* will disallow this recursive call. It will display the message: "I cannot show termination under this edit." Note that there is no algorithm to decide termination in all cases. Inevitably therefore, there are terminating programs that cannot be produced using *CYNTHIA*. However, *CYNTHIA* works fine in the majority of cases.

CHANGE TERM

```
int list -> int

fun sum x = [ 0 ];
```

Change an expression at the current point in the definition. Here, `x` was entered to replace `0`.

```
int list -> int

fun sum x = x;
```

CHANGE TYPE

```
[ int list ] -> int

fun sum nil = 0
|   sum (x::xs) =  sum xs

int tree -> int

fun sum (leaf n) = 0
|   sum (node(x,xs,e1)) = sum xs;
```

Change the top-level type of a definition (including local functions). Here, we have changed the type of sum's argument to an integer tree. Note that the pattern by which the function is defined has been changed to a pattern for trees. In general, this procedure may introduce or remove variables. Note also that a recursive call `sum e1` will be made available (see bottom right window in *CYNTHIA*).

MAKE PATTERN

```
int list -> int

fun sum [ x ] = 0;
```

For a variable of some type, provide cases for each constructor in that type. New variables may be introduced. Can be applied iteratively to build up complex patterns.

```
int list -> int

fun sum nil = 0
|   sum (x::xs) = 0;
```

RENAME

```
int list -> int

fun sum nil = 0
|   sum (x::[ xs ]) =  sum xs;
```

Rename a variable or function name. User provides the new name. Variable names are as in SML. Invalid names will be detected at the time of the command application.

```
int list -> int
fun sum nil = 0
|   sum (x::z) = sum z;
```

UNDO

Select undo from the main EDIT menu. Undo returns the program state to before the most recent editing command was applied. Only a history of one command is provided.

B.5 Feedback Provided by *CYNTHIA*

B.5.1 Syntax Errors

It is possible to write syntactically invalid definitions in *CYNTHIA*. This is because some editing commands will remove objects from one part of the definition but not elsewhere in the definition. For example, suppose you applied remove curried argument to the following function (the boxed expression denotes where the command has been applied. It is not a *CYNTHIA* colouring):

```
fun add x [y] = x + y;
```

After this command has been applied `y` will still be present on the right-hand side of the equality. However, `y` has not been declared so there is a syntax error. *CYNTHIA* will notify the user of such syntax errors by colouring the offending term(s) in GREEN:

```
int-> int
fun add x = x+y;
```

The correction of these sorts of errors can be done directly or indirectly:

- Directly - change the expression `y`
- Indirectly - apply some other command (e.g. add `y` as an additional argument)

B.5.2 Type Errors

In a similar way to syntax errors, type errors are indicated in PINK. Note that a little more care is necessary when dealing with type errors, for the coloured expression may not always be the source of the type error.

The following *CYNTHIA* definition contains a type error.

```
int-> int-> int list
fun add x y = x+y;
```

`x+y` is not of type `int list` and so is highlighted.

B.5.3 Error Messages

`Incorrect syntax:`

The expression you are editing has a syntax error.

`Wrong number of arguments:`

You have given a function an incorrect number of (curried) arguments. Note: be aware of giving non-curried arguments where curried arguments are required. E.g. if `f` takes two arguments and you type `f (a,b)`, *CYNTHIA* will interpret this as a single argument of type pair.

`Variable is already in use:`

You have tried to introduce a variable that is already being used. In ML, variables are local to each case of the definition. This feature is not implemented yet in *CYNTHIA*, so variables from one case cannot be used in another case.

`Constructors cannot be used as variable names:`

You have tried to use a constructor name (such as `nil` or `leaf`) as a variable name.

`Reserved words cannot be used as variable names:`

The list of reserved words in *CYNTHIA* is slightly different to those in ML.

`Built-in functions cannot be used as variable names:`

The list of built-in functions is slightly different to those in ML.

`CHANGE TERM cannot be used to alter recursive calls:`

To do this you must use ADD RECURSIVE CALL.

This tutorial was originally posted on the WWW for use by the students. The Web version can be found at http://www.dai.ed.ac.uk/students/jonathw/cynthia/tutorial.html.

Appendix C

Questionnaire to Assess Students' Difficulties

Questionnaire

This questionnaire is intended to survey the kinds of programming errors encountered when using the language ML. The first section is intended to be specific to today's session. Please answer based on your experiences from TODAY ONLY. The second section contains more general questions.

Section One

The following list categorises certain types of errors that can occur in ML. Please tick any that you came across.

[ ] syntax was wrong e.g. missing parentheses
[ ] there was a type mismatch
[ ] the program got caught in a loop or failed to stop
[ ] other (please state):

For each of these categories of error, please indicate how often you encountered the problem:

syntax error:

[ ] never
[ ] hardly ever
[ ] frequently
[ ] very often

type mismatch:

[ ] never

[ ] hardly ever
[ ] frequently
[ ] very often

looping error:

[ ] never
[ ] hardly ever
[ ] frequently
[ ] very often

couldn't identify the reason for the error:

[ ] never
[ ] hardly ever
[ ] frequently
[ ] very often

How easy was it to correct the errors?

syntax error:

[ ] very easy
[ ] relatively easy
[ ] neither easy nor difficult
[ ] relatively difficult
[ ] very difficult

type mismatch:

[ ] very easy
[ ] relatively easy
[ ] neither easy nor difficult
[ ] relatively difficult
[ ] very difficult

looping error:

[ ] very easy
[ ] relatively easy
[ ] neither easy nor difficult
[ ] relatively difficult
[ ] very difficult

Section Two

How helpful do you think the ML compiler is in identifying errors of the kind mentioned in section one?

[ ] Not at all helpful

[ ] Only slightly helpful
[ ] Moderately helpful
[ ] Mostly helpful
[ ] Extremely helpful

What proportion of your programming time is spent correcting these kind of errors?

[ ] Almost none
[ ] A small amount
[ ] A fair amount
[ ] A large amount
[ ] A very large amount

Please indicate your general level of computer programming experience:

[ ] Never programmed before
[ ] Minimal programming
[ ] Good amount of programming
[ ] Experienced programmer

What computer languages have you programmed in before?

Below this line, please add any comments you have on the ML compiler or the activity of writing ML programs

Appendix D

Grammar Supported by *CYNTHIA*

This follows the notation in the Definition of SML [Milner *et al.* 90].

atexp	::=	*scon*
		var
		con
		()
		$(exp_1, \ldots, exp_n)$
		$[exp_1, \ldots, exp_n]$
		`let` *dec* `in` *exp* `end`
		(*exp*)
appexp	::=	*atexp*
		appexp atexp
infexp	::=	*appexp*
		$infexp_1$ *id* $infexp_2$
exp	::=	*infexp*
		exp_1 `andalso` exp_2
		exp_1 `orelse` exp_2
		`if` exp_1 `then` exp_2 `else` exp_3
		`case` *exp* `of` *match*
		`fn` *match*
match	::=	*mrule* < | *match* >
mrule	::=	*pat* `=>` *exp*
dec	::=	`val` *valbind*
		`fun` *fvalbind*

$$
\begin{array}{lcl}
valbind & ::= & vpat \texttt{ = } exp \\
\\
fvalbind & ::= & \quad var\ atpat_{11} \ldots atpat_{1n} \texttt{ = } exp_1 \\
 & & \texttt{| } var\ atpat_{21} \ldots atpat_{2n} \texttt{ = } exp_2 \\
 & & \texttt{| } \ldots \quad \ldots \\
 & & \texttt{| } var\ atpat_{m1} \ldots atpat_{mn} \texttt{ = } exp_m \\
\\
atpat & ::= & scon \\
 & & var \\
 & & con \\
 & & () \\
 & & (pat_1, \ldots, pat_n) \\
 & & (pat) \\
\\
pat & ::= & atpat \\
 & & con\ atpat \\
 & & pat_1\ con\ pat_2 \\
\\
vpat & ::= & var \\
 & & () \\
 & & (vpat_1, vpat_2) \\
\\
ty & ::= & tyvar \\
 & & ty\ tycon \\
 & & ty_1 \texttt{ * } \ldots \texttt{ * } ty_n \\
 & & ty \texttt{ -> } ty' \\
 & & (ty)
\end{array}
$$

The reader should consult [Milner *et al.* 90] for a full explanation of the notation used in the grammar. Only a minimal description is given here. *ty* is a grammar for types. *exp* is a grammar for expressions.

scon is a special constant which is either an integer, a string or a real. *id* is an identifier. An identifier is either alphanumeric (any sequence of letters, digits, primes and underscores starting with a letter of prime) or symbolic (any non-empty sequence of reserved symbols such as `!` and `<`. *var* is a variable which is any alphanumeric identifier. A type variable *tyvar* is any alphanumeric identifier starting with a prime. *con* and *tycon* are value and type constructors respectively. The brackets $< >$ enclose optional phrases. Alternative forms for each phrase class are in decreasing order of preference.

CYNTHIA does not support all definitions that match the above grammar. Functions that are not Walther Recursive are not accepted. Tuples are allowed only in predefined type constructors, e.g. `node(n,x,y)`. Otherwise, they must be written as pairs of pairs. Certain destructor-style definitions are not allowed, e.g.:

```
fun f x = if x = nil then 0
          else 1 + f (tl x);
```

This is Walther Recursive but *CYNTHIA* currently only allows recursion based on constructor-style definitions. This is an implementation oversight.

I have introduced an additional phrase class *vpat*. This is a restriction of *atpat* defined since *CYNTHIA* only allows a restricted form of pattern matching on the left-hand side of the equality of a `let val` statement. SML allows definitions such as:

```
let val (x::y) = g z in x end;
```

This is not allowed in *CYNTHIA*.

Appendix E

SML of New Jersey Error Messages (v.0.93)

syntax error: syntax error in input.

NONfix pattern required: a constructor is being used that has been declared as an infix but the keyword `op` has not been included.

unclosed string: quotes missed off the start of end of an expression.

nonfix identifier required: an identifier is being used that has been declared as an infix but the keyword `op` has not been included.

unbound variable or constructor: expression contains a variable or a constructor that has not been declared in the current context.

operator not a function: the value used in an operator position is not a function. For example, in `3 true`, needs to be a function.

clauses don't all have same number of patterns: associated with a definition of a function using pattern matching where the function is given a different number of arguments in different clauses.

duplicate variable in pattern: the same variable cannot be used more than once in a pattern.

can't find function symbol: often caused by bad parentheses. Tried to define a function without a function symbol. For instance, in `fun (f x) = 3;`, the parentheses should be removed.

constructor used without argument: missed an argument to a constructor.

tycon mismatch: the actual type of an expression does not agree with the type expected from previous type inferences.

overloaded variable: some functions can be defined over more than one type. An example is + which may be over integers or reals for instance. This error message is displayed if the type over which the function is defined is ambiguous in the current

context. Can usually be solved by including an explicit type declaration.

match redundant: more than one pattern matches some expression.

match non-exhaustive: there is some expression that is not matched by any of the patterns given.

uncaught Match exception: this is a run-time error flagged when the compiler is asked to evaluate an expression that is not matched by any of the patterns in the current contezt.

I have only included messages referred to in Chapter 2. For others, see the SML documentation (at http://cm.bell-labs.com/cm/cs/what/smlnj/index.html).

Appendix F

The Recursion Editor Commands

Non-Recursive Commands	
Remove dependency (rd)	Removes an argument from a function in the body of a clause
Remove parameter (rp)	Removes all occurrences of a parameter in a function
Add parameter (ap)	Adds a parameter to a function
Move Argument (ma)	Swaps the order of two arguments in a function
Rename	Instantiates meta-symbols or variables to the given name
Insert Function (if)	Introduces some function into the body of a function
Multiply Cases (mc)	Introduces the number of specified conditional cases
Recursive Commands	
Add Recursion Argument (rap)	Adds a recursive argument to a function
Add Constructor Recursion Arguments (rac)	Adds recursive arguments to a given constructor function
Multiply Base Constructor (mbc)	Adds a specified number of base cases
Multiply Step Constructor (rmc)	Adds a specified number of step cases
Insert Parameter Procedure (ripf)	Introduces a given procedure as an extra argument to the recursive call of a function
Insert Recursive Procedure (rim)	Introduces mutual recursion
Insert Constructor Function (ricf)	Changes the constructor function of a recursion e.g. to change the step size

Note the difference between **rd** and **rp**. **rd** removes just one occurence of a variable or function, so changing

$$f(s(N), Y) = g(N, Y, f(N, Y)) \tag{F.1}$$

to

$$f(s(N), Y) = g(N, f(N, Y))$$

whereas **rp** removes all occurrences of some parameter changing (F.1) for example to

$$f(s(N)) = g(N, f(N))$$

For further details on any of these functions see [Bundy *et al.* 91]. Note that "Insert Function" is called "Insert Procedure" in that paper.

Index